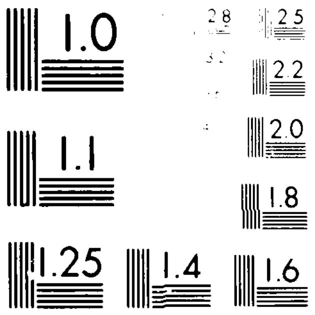
AD-A110 366   UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT O--ETC F/G 9/2
A STUDY OF FACTORS AFFECTING SOFTWARE TESTING PERFORMANCE AND C--ETC(U)
JUN 80   J L BAHR

UNCLASSIFIED

NL

1 of 3
AD A
110366


1.0
2.8
2.5
3.2
2.2
2.0
1.1
1.8
1.25
1.4
1.6

[illegible]

LEVEL

1

USC

GRADUATE SCHOOL OF BUSINESS ADMINISTRATION

AND

SCHOOL OF BUSINESS

UNIVERSITY OF SOUTHERN CALIFORNIA

A STUDY OF FACTORS AFFECTING SOFTWARE TESTING PERFORMANCE AND COMPUTER PROGRAM RELIABILITY GROWTH

by
Jeffrey Louis Bahr
DEPARTMENT OF MANAGEMENT AND POLICY SCIENCES
UNIVERSITY OF SOUTHERN CALIFORNIA

June 1980

This research was supported in part by the Office of Naval Research under Contract N00014-75-C-0733, Task No. 042-323, Code 434. Reproduction in whole or part is permitted for any purpose of the United States Government.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Since the introduction of computing systems three decades ago, software development has grown increasingly in importance and complexity. Today the expense incurred by users in producing and maintaining programs exceeds ten billion dollars (Phister, 1976) and constitutes a large fraction of some corporate budgets (Dorn, 1978). Joint revenues of independent software suppliers exceeds one billion dollars, while revenues of computer manufacturers is approximately twice that (International Data Corporation, 1978). The increases in software-related expense far outstrip the growth in hardware costs. While the labor-intensive nature of software production has contributed to its increasing cost, competitive pressure and increased manufacturing automation have reduced the cost of hardware elements. It is for this and other reasons that Boehm (1973) predicted that software costs will constitute 90 percent of total system costs by 1985.

As user needs become satisfied, software systems are designed to increasingly demanding specifications. This rapid growth in the number and complexity of desired systems demands equivalent increases in the number of qualified software specialists. More importantly, it dictates a maturity in software development methodology. The current state of "software engineering" does not exhibit this degree of maturity, nor does any significant proportion of software workers employ those tools and techniques which most influence the quality of software systems.

The Software Problem

Researchers and practitioners in the information sciences refer to the "software problem" to indicate the special characteristics of software as an artifact, and the ways in which those characteristics affect its production. Wegner (1978) enumerated the special properties of software that distinguish the process of its manufacture from that of other engineering projects:

- Large software products generally support a greater variety of functions than conventionally engineered goods.
- There is an enormous variety of correct implementations for a large software system, from which one must be "selected."
- A large software system must be constructed so as to mitigate the cost and difficulty of its inevitable modification.
- Software development milestones are difficult to establish, hence project progress is hard to assess.

Since software partly resides in the realm of ideas, the complexity of its implementation is seldom as obvious as that of physical artifacts. Moreover, there is some evidence that large software systems exhibit properties substantially different from small software systems, not just differences in scale (Boehm, McClean, & Urfrig, 1975). Because of this, prototyping of software systems does not generate as much useful information as a physical model which may be developed preceding the construction of physical goods.

Society's appreciation of the software problem has been constantly diluted by exposure to advances in the total computing system. That is, increasing complexity in large software systems has been partially masked by the apparent ease with which hardware components have been expanded in the last two decades. In attempting to utilize increased capacities for information processing, a natural demand has been created for that complementary good which controls the execution of hardware elements. A large percentage of the gains in hardware, however, have come about with less additional complexity than would be necessary to increase software system size. Reductions in cost and increases in hardware have been substantially effected by methods which are only linearly more complex than older ones. These methods include miniturization of components, and increases in resource volume and density without an associated increase in interface complexity. Conversely, advances and improvements in software have come about primarily through the layering of capability after capability, which, at each step, gently increases system complexity.

Management of software production has been hindered by the fact that the software development process generates inherently less visible evidence of progress than other development projects. Walker (1978) attributed this to the fact that the primary activity in software development is that of communication, and that without filtering mechanisms a substantial amount of noise is injected in the messages which become components of a software system. Visibility of user needs is also often obscured. Users normally increase their expectation when they are able to state their needs more explicitly in terms of current system service. These needs may not be closely examined; because of its intangibility, however, software is expected to be sufficiently malleable to accommodate them. Hence, maintenance is not always subjected to the same degree of feasibility analysis as would accompany a modification request of a physical good.

Modern Development Methods

It is the current view of researchers in software engineering that aspects of the "software problem" can be minimized through the use of disciplined methods of specification, design, and coding, set within a management framework that permits the planning, measurement, and control of development activities. The most important factor influencing development and maintenance research has been the rediscovery of the life-cycle metaphor applicable to software systems (Wegner, 1978; Zeklowitz, 1978). The life-cycle approach to software development recognizes the various stages through which the software system can be viewed as progressing. With each phase of a software system's life is associated a set of costs and benefits incurred in its development, use, and maintenance.

Early software development efforts focused on the reduction of development costs and duration. Experience in the last decade with systems that partially or completely failed to meet specifications and budgetary constraints forced an expansion of this myopic view. A series of regimens has been proposed that promote the traceability of requirements through the succession of representations of the system, while increasing one's ability to understand a software system's structure and function. These modern methodologies include variants of structured programming (Dahl, Dijkstra, & Hoare, 1972), structured design (Stevens, Myers, & Constantine, 1974), and structured analysis (Ross & Schoman, 1977). Enlightened project management can exploit these methodologies while instituting measurement and validation techniques such as peer review and formal testing, as well as the normal management reporting methods.

Software development can be viewed as a process that translates specifications into a working system through a series of transformations that provide intermediate representations. In making operational the system components that satisfy the general set of functional, economic, and performance constraints, a succession of decisions has to be budgeted among development team participants.

Validation is necessary to assess the effects of those decisions before the system becomes too costly to restructure. Prior to the development of a system code this validation normally takes the form of reviews in which users, developers, and project managers cooperate to determine conformance to the project's plans and goals. The most expensive and time-consuming form of validation, _testing_, currently and in the past, has occurred after the design has been translated into programming language. Alberts (1976) has estimated that up to 50 percent of development cost is incurred during the testing of software system elements, whereas up to 90 percent of life-cycle costs involve maintenance to correct errors and rewrite system codes to meet new requirements.

The Role of Validation

Validation Research Areas

Areas for research in system validation may be categorized in a variety of ways, including: by development phase addressed, manual methods vs. automated tools, or the degree to which the results of other disciplines are employed. Another useful dichotomy is suggested by the alternative views of software as a static document versus software as a dynamic system. Hence the validation of system code can take two general forms:

- System verification, which attempts to prove the correctness of the implementation by formal methods, and
- System testing, which involves the application of test data sets that exercise a software system in a manner representative of future and current use.

One important area in testing research is software reliability theory, in which models are proposed that serve to approximate or predict the frequency and composition of faults that may occur during system operation. Because testing is costly, reliability theory gives one a means of maximizing expected utility by indicating the level of testing necessary to trade off optimally the cost of further quality assurance measures with the cost of improper system execution. The tradeoffs differ from system to system and depend, for example, upon whether a system's errors will result in financial discomfort or loss of human life. An excellent overview of software reliability theory is given by Schick and Wolverton (1978).

Early Validation

"Early validation" refers to software development practices that attempt to enhance the fidelity of translation from problem statement to software system design. Proponents of early validation are justified in believing that software system reliability must be maintained through the phases of system construction that precede the programming stage.

Static analysis methods involve the examination of documents representing various aspects of the system. Analysis of requirements statements, formal specifications, and design documents is normally static because these representations are seldom machine readable and almost never executable.

The importance of requirements analysis has become a major theme in the literature of software engineering for large systems (Bell & Thayer, 1976; Ross, 1977a). Requirements enumerate the needs of the system's client(s) by communicating a desire for function within some performance context. Analysis of requirements implies checking for the desired properties such as consistency, completeness, lack of ambiguity, and feasibility.

Formal automated systems for describing and analyzing requirement sets are under development. Systems currently exist which automate many of the informal checks for properties desirable in specifications. The ISDOS system (Teichroew & Hershey, 1977) and REVS (Davis & Vick, 1977), for example, check to ensure the uniformity of data definition and use and consistency of views of module interface, while providing reports facilitating manual validation of system descriptions.

A variety of language forms and media are available for design representation (Ross, 1977b; Caine & Kent, 1975; Stay, 1976). Like requirements analyses, the validation of design documentation is normally informal and restricted to review and discussion among the development team. Such reviews include the tracing of requirements to design, analysis of control logic, critical comparisons of algorithms, and other analyses to verify consistency, necessity, sufficiency, and correctness. Automated analysis techniques are currently limited to tools that check interface consistency or simulate structured models of system execution (Ramamoorthy & Ho, 1975).

Validation by Symbolic Verification

Program cerification refers to the use of mathematical proof techniques to validate program correctness. A program proof requires a goal proposition that indicates the relationships that should hold between program variables at the conclusion of the program's execution. This goal proposition is normally termed an output assertion; assumptions regarding system states prior to execution (e.g., implied relationships between subroutine parameters) are reflected by input assertions.

A program is (totally) correct if it terminates for all valid input, in a state for which the output assertion holds. A proof of total correctness must treat program statements as theorems that relate statement preconditions to postconditions. A case-by-case analysis is usually necessary to treat the proof subsequences resulting from decision points in the program. These notions are equally important in top-down verification (constructive proof validating output assertion through a series of implications originating at the input assertion) and bottom-up verification (showing that the output assertion is true for an input domain, part of which conforms to input assertions).

On the validation continuum between program verification and test case application, reside the techniques associated with symbolic evaluation. Symbolic execution of a program can be performed in the same way as actual execution: the state of program progress is recorded by the values of variables and current instruction. The difference in symbolic and actual execution is the form of value assigned to variables and the evaluation of conditions dictating the execution path. The symbolic evaluation of a path is carried out by evaluating the sequence of, say, assignment statements occurring in the path. Each assignment statement provides a new value for the target variable (left-hand side) through substitution of the current symbolic values for each variable in the source expression (right-hand side).

The branching conditions that control the selection of execution paths (say, if an IF or DO-WHILE) can be symbolically evaluated to form symbolic predicates. The sequence of symbolic predicates associated with a path can be generated by assuming a truth value for each decision point during program execution. This symbolic system of predicates for a path can be used to assist the tester in constructing test data. If the system of predicates for a path is unsolvable, then the path is infeasible; otherwise the solution describes the subset of the input domain (uninitialized variables and system parameters) that invokes execution of that path.

Validation of System Testing

Testing involves the application of sets of data in a controlled environment to assess the reliability of a software system and identify errors, if any, in the system representation that may be corrected. Testing goals usually include the determination of whether the system meets functional and performance specifications associated with user requirements. Requirements-oriented goals are not normally considered a sufficient set of criteria for system testing, for they are seldom sufficiently detailed to indicate the test data that prove conformance. Test criteria of a larger scope are also necessary to insure that the system behaves correctly under conditions not anticipated by system users.

Researchers in testing theory have attempted to identify the characteristics of testing criteria that could identify test data whose application could insure system reliability to the same degree provided by formal proofs of correctness (Goodenough & Gerhart, 1975). These system validation criteria normally arise from one of two sources:

- A set of specifications that detail conditions for success as dictated by the requirements document.
- Criteria composed by analyzing program structure to discover conditions that lead to successful program execution.

Data generators exist to produce test data subject to each of these two approaches (Stucki, 1977).

Consider a test data set, S, generated to affirm conformance to specifications, and test data set P, generated by analyzing program structure. A mechanism for exploiting these two forms of test data generation may analyze the union of the two data sets, S and P, to reduce their number by eliminating redundant data while providing useful information about test data inconsistency and the manner in which each of the two test data sets was incomplete.

A related problem occurs when testing for regression errors. Regression errors are those errors introduced into the system during system modification. Brooks (1972) provides evidence that a substantial percentage of maintenance activities introduce errors that did not previously exist. In retesting the system after maintenance, it is theoretically sufficient to apply a limited set of test data: those test data that reaffirm the correctness of the modified components and those system elements affected by the modified elements.

The Role of Errors in Software Research

Computer-related studies may be loosely dichotomized into those subdisciplines dealing with function and those concerned with fidelity and form. In the former class can be included the application areas (operations research, management information systems), the algorithmic areas (numerical analysis, time and space efficiency analysis), and the special systems areas (operating systems, data base management). Fields of study in the second category include those disciplines that recognize and contend with the difficulty of producing automata that conform to any desired function; in short, they share a view of software that recognizes the error as a primitive component of the development process. Because the difficulties of proper design and implementation are apparently application-independent, each approach to these difficulties exhibits a characteristic generality even though each differs widely in method. The following taxonomy is proposed to describe these approaches:

- Error Control: the application of numerical analysis to bound the inaccuracy of finite machines.
- Error Prevention Methodologies: development procedures like structured analysis, design, and programming that prescribe means with which to facilitate the translation of problem statement to coded solution.
- Error Prevention Tools: programming language and support system refinements that reduce the complexity of implementation.
- Error Detection Methodologies: systematized debugging procedures like desk-checking and structured walkthroughs that increase the likelihood of finding misconstructions.
- Error Detection Tools: software tools like cross-referencers, static analyzers, and dumps that point out syntactic and procedural anomalies.
- Fault-Tolerance: the study and design of hardware or software systems that recover from failure with minimum damage.
- Proof-of-Correctness: the study and design of methods for constructively assessing the conformance of a procedure to its specifications.
- Failure-Inducing Methods: test data selection methodologies like path-based and specification-driven approaches in which sets of program inputs are chosen to "provoke" errors to evidence themselves.

- Failure Behavior Modeling: the branch of Software Reliability Theory in which the phenomenon of program failure is viewed as stochastic and conforming, in the aggregate, to specified distributions.

All of these approaches are associated with fields of study and vigorously coexist as complementary solutions to problems of anticipating, recognizing, explaining, predicting, and correcting for program malfunction.

A Comparison of Software Reliability Theory and Modern Testing Theory

The areas of research listed in the previous section have, in general, evolved from a common base of thought. Each of the disciplines associated with these approaches has benefited from concepts and advances offered by another. However, the areas of Modern Testing Theory (Failure-Induced Methods) and Software Reliability Theory (Failure Behavior Modeling) appear as disjoint a pair as any two research areas in this collection.

It can be surmised that the differences evident between these approaches have their roots in the training and motivations of the relatively disparate research groups that publish in each area. Modern testing theorists and practitioners have invariably received training in the computer sciences, view programs as manifestations with mathematical structure, and consider the most effective and efficient means of "flushing out" what errors remain. By comparison, the majority of Software Reliability Theorists have a background in statistics, view programs as stochastic event generators, and concern themselves with modeling the sources of nondeterminism and probability distributions that describe their behavior. Table 1 exhibits a synopsis of the methodological and philosophical differences separating these disciplines.

An Overview of this Research

In this study we report the results of an experiment conducted to determine the many influences which affect the process of software testing. To motivate aspects of testing methodology chosen for experimental analysis, a review of the literature in Testing Theory and Software Reliability Theory is given in Chapters II and III. The material in Chapter II outlines the testing tools and approaches that have been proposed, employed, and studied by researchers and practitioners engaged in software validation. Chapter III reviews the most important models proposed to describe the failure behavior of software systems under test. Chapters IV and V describe an experiment designed to investigate the claims and assumptions made in the literature. In the final chapter we will report the significance of our findings and suggest how they should affect the future choice of research avenues in the field of software validation research.

Table 1

A Comparison of Modern Testing and Software Reliability Disciplines

	Modern Testing Theory	Software Reliability Theory
Principal activity	Finding errors	Estimating error content
View of errors	Deterministic	Probabilistic
Eventual aim	Assurance of error elimination	Proper reliability assessment
Published successes	In relatively small software systems	In relatively large software systems
Typical disciplines of investigators	Computer science	Statistics
Typical publication topics	Theory and practice of test data selection	Models of software failure behavior
Principal publication vehicle	Software Engineering Literature	

CHAPTER II

THEORY TESTING AND PRACTICE

Introduction

A program P may, in theory, be proved correct in one of two ways: by formal verification or exhaustive testing. P effectively computes a function f by correctly mapping elements of f's input domain D into the proper element of f's output range. Formal verification normally involves a case analysis of the program P to prove symbolically that the collection of partial functions composing f is faithfully represented in the structure of the program. Although research proceeds vigorously on program verification, there is little hope that large or complex programs will admit of proof in the near future (Huang, 1975).

Exhaustive testing is also out of the question for all but trivial programs, because even one 32-bit integer input requires the application of 2^{32} test cases (and inspection of results) to prove program correctness. One obvious simplification in the test procedure follows from the consideration of program structure.

Program execution is a matter of successive instructional interpretation (as mirrored by a path through a program flowchart). A substantial reduction in the size of the test data set is expected by applying one test data set for each unique flow through the program.

Figure 1 shows a sample flowchart associated with a program P, composed of functional instructions $S_1,\cdots,S_6$ and decision points D_1, D_2, and D_3. The input domain I of this program may be partitioned into subsets $I_1,I_2,\cdots,I_n$ where every input data point in I_j leads to the same execution sequence of the functional instructions $S_{i1},S_{i2},\cdots$; that is, I is partitioned into subdomains, where each member of a subdomain leads to the same path through the flowchart.

A more compact form of representation is useful for demonstrating path analysis. The control graph (Figure 2) represents a program's uninterrupted sequence of functional statements (termed segments) as nodes, and alternatives for execution flow as arcs. The control graph arcs can be labeled by the conditions that must be satisfied for an arc to be selected, as the c_i and $\bar{c}_i$ represent the true-valued and false-valued outcomes of the decision corresponding to D_i in the flowchart.

Testing involves the selection of a subset T of the input domain, the application of each data point t in T to the program and the determination that the program output P(t) is an acceptable result, i.e., that P(t) is an effective computation of f(t). In a seminal paper, Goodenough and Gerhart (1975) attempted to define the conditions under which testing is the practical equivalent of a formal correctness proof. Howden (1976) elucidated these results in the following way: Consider a test strategy H, associated with a test criterion C_H. H is a procedure for selecting a subset T of a program's input domain so that the $t \in T$ individually or collectively satisfy the test criterion C_H.

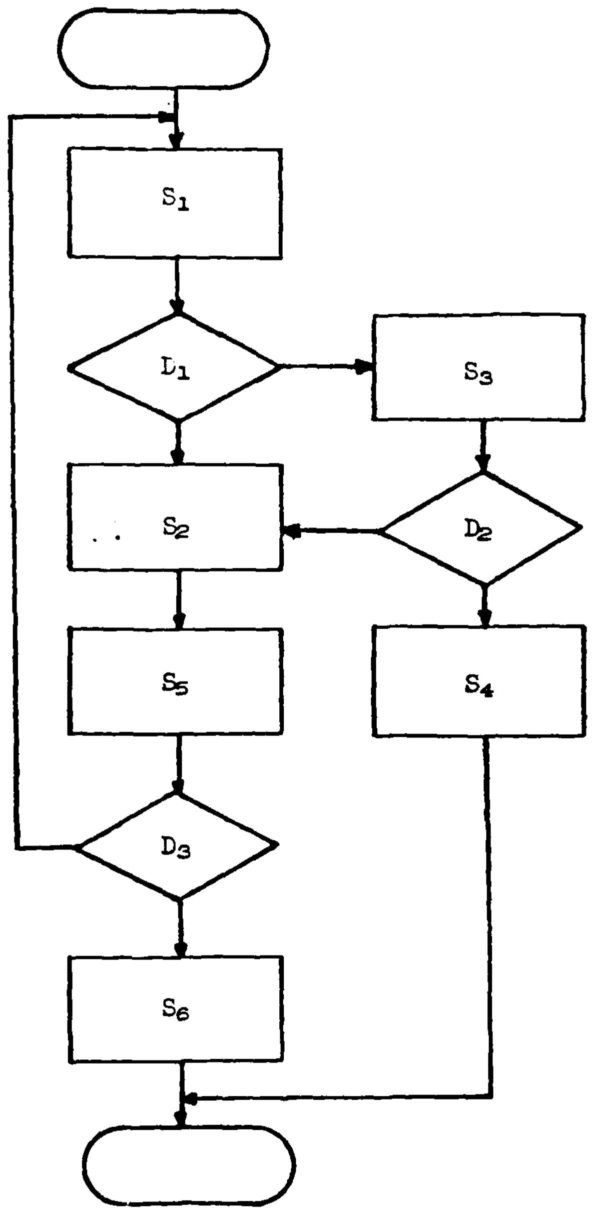


Figure 1. A program flowchart

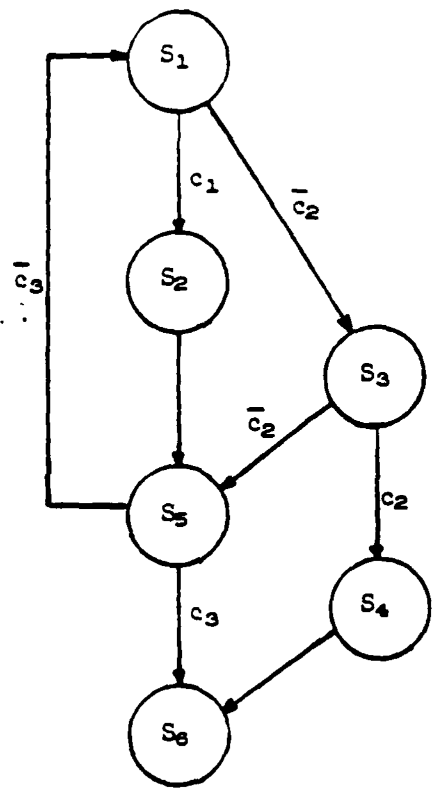


Figure 2. A program control graph

T^* is a reliable test set if the successful execution of every $t \in T^*$ implies the correctness of P. Because no strategy exists that can generate T^* for an arbitrary program (this is undecidable), work proceeds on identifying particular classes of programs and program errors for which some test criterion is reliable (Howden, 1976; Gerhart & Yelowitz, 1976). Howden (1976) identified two general classes of program error by indicating the reliability of path-testing in discovery of each error class. A path-testing strategy is a procedure for selecting one data point from each subset D_j of the input domain, that is, the application of one test case for each unique path through the program. Such a strategy is reliable for discovering

- computation errors, in which a subject program P has an identical structure to the correct program P^* (same paths, same partitions D_j) but always yields a differing result for some path;
- domain and subcase errors, in which both a correct program P have the same collection of paths, but a differing partitioning of the input domain associated with the paths, and a differing selection of paths, which yields differing results (Howden, 1976).

The results from the theory of testing have little practical applicability, but do provide bounds on the kinds of errors that can be discovered (at all) by testing criteria less comprehensive than path testing. Like the problem classes proposed by computability theorists, modern testing theorists have attempted to prove the inherent difficulty of various kinds of errors.

Testing Approaches and Results

Structural Testing

The identification of paths is an integral part of every formal testing procedure. Prior to the realization that formal testing was worth its cost, programmers constructed test data sets in hopes of discovering errors, thus ensuring proper program execution for those data deemed representative of real system use. The obvious opportunity for misjudgment and resulting proliferation of errors "in the field" has led many organizations to formalize testing by creating test data that might affirm conformance to the explicit functional and performance requirements of the system under test. In a study of six large programs, Howden (1978a) found that functional testing uncovered more errors than test criteria based solely upon program structure.

Functional testing is effective but can fail to detect many errors that may be detected by more methodological approaches (Ramamoorthy & Ho, 1975). Structural testing provides a complementary method of program validation by approaching the problem by analysis of the program. Path testing is often impossible because the number of paths through a program can be very large. In fact, if a cycle with no bound on iteration exists within a program, the number of paths may be unbounded. There are three common test criteria that attempt to exercise relevant aspects of a program without resorting to full path testing: segment testing, branch testing, and selective path testing. Each testing approach attempts to identify a set of program paths whose successful execution will increase the confidence in program correctness.

Segment testing involves the construction of test cases to ensure that every statement in the program is executed at least once. If a program statement is seriously misspecified, segment testing may identify a program error irrespective of the execution path in which the erroneous statement is included. Segment testing derives its name from the concept of a segment: a series of program instructions with a consecutive execution sequence. A reduced control graph for a program normally contains vertices associated with program segments. The first statement in a segment is the object of one or more branching instructions (designated by arcs). The last statement either leads directly to a decisional statement or <u>is</u> a decisional statement (depending upon the desire to implicitly or explicitly represent branching statements in the control graph).

Figures 3 and 4 show an abstract program and control graph in which vertices have been employed to represent segments of sequential instructions. The paths (a,b,d,e,f,j) and (a,b,g,i,f,j) in graph G provide a covering of the vertices of the control graph G and so ensure the execution of every segment. An alternative testing strategy involves the application of test data to exercise every decision branch in a program, hence every edge in the control graph. The minimal number of test cases required to ensure edge covering serves as an upper bound on the number for vertex covering, for if every intersegment branch is taken, every segment must be invoked. The path set {(a,b,d,e,f,j),(a,b,g,e,f,j),(a,b,g,i,f,j),(a,b,d,e,f,b,g,i,f,j)} is one edge covering for G.

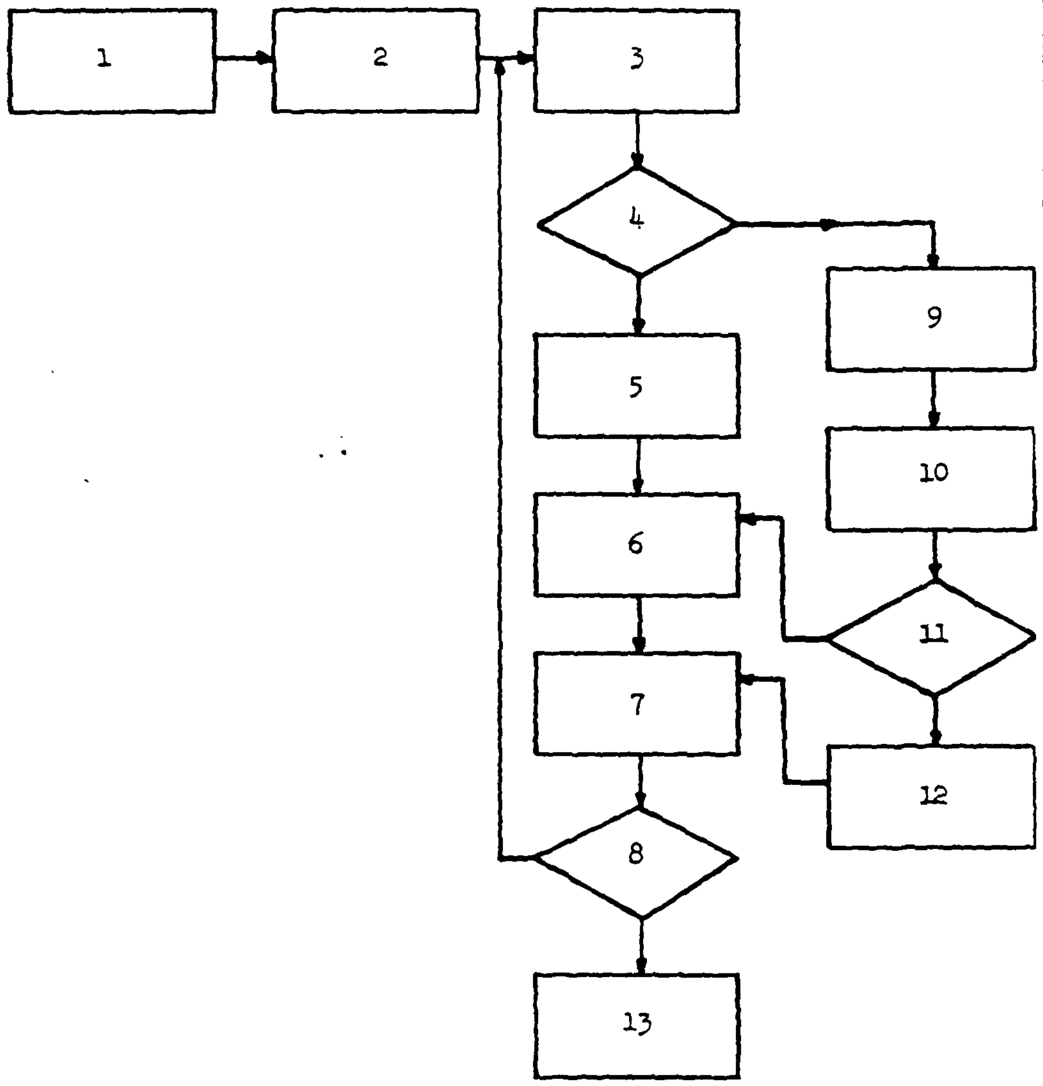
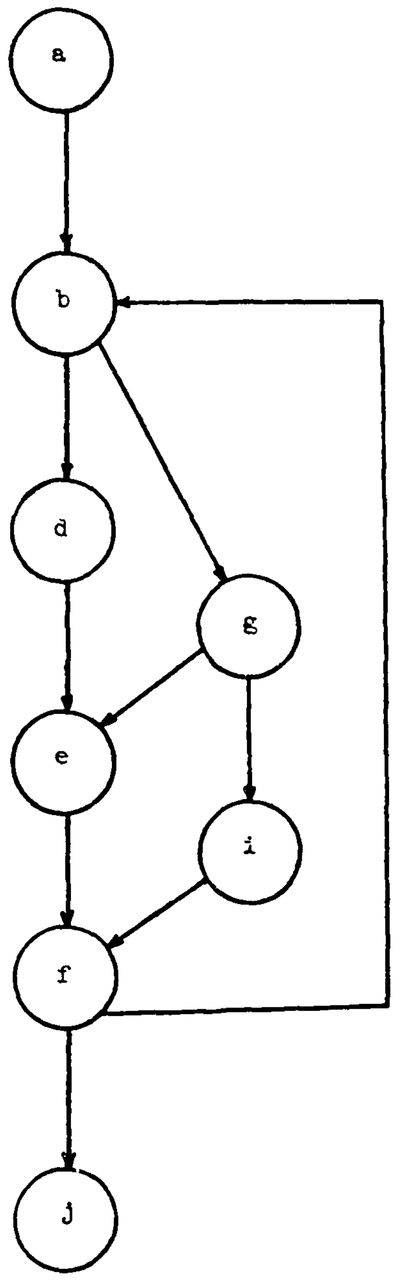


Figure 3. Program flow chart

SEGMENTS	NODES
1,2	a
3,4	b
5	d
6	e
7,8	f
9,10,11	g
12	i
13	j



$$\begin{tikzcd}
& a \arrow[d] & \\
& b \arrow[d] \arrow[rd] & \\
& d \arrow[d] & g \arrow[ld] \arrow[d] \\
& e \arrow[d] & i \arrow[ld] \\
& f \arrow[d] \arrow[uuu, bend right=60] & \\
& j &
\end{tikzcd}$$

Figure 4. Reduced control graph G

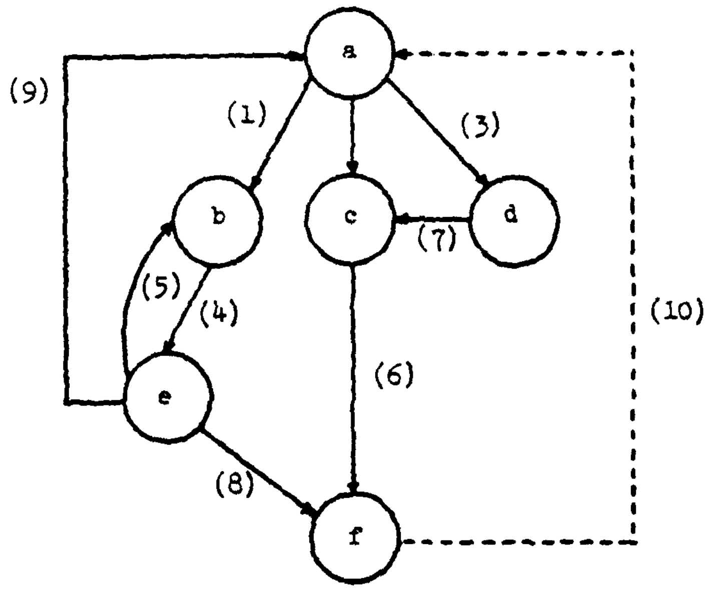
Reported results with segment and branch testing effectiveness have varied, depending upon the sophistication of testing procedures that were replaced by these formal methods. In one aerospace application, branch testing purportedly eliminated 90 percent of program faults (Brown, 1975). In another study of development costs for a command and control system, Alberts (1976) concluded that automated tools employing branch testing caught 60-100 percent of all program errors two to five months earlier than they would otherwise have been detected. Very few results have been published indicating the effectiveness of vertex covering. Fisher (1977) employed segment testing (vertex-covering) as a retest criterion to select test cases to reapply after program modification. In a small study comparing a vertex-covering test data set to an *expanded edge-covering set*, Brown and Lipow (1975) found the latter much more representative of a variety of assumptions regarding input data distributions. Segment testing is, however, nearly always superior to intuition. For example, one study of typical testing practice in a large software organization revealed that on the average only one-third of all program statements were being exercised by existing test data sets (Stucki, 1978).

Bounds on the Test Set Cardinality

It would seem that in some situations the tradeoffs between the cost of testing and desired program reliability would lead one to opt for a vertex covering approach. One factor in the choice of edge over vertex covering test criteria is the additional resource required by the latter, depending upon the relative sizes of their minimal test sets. A bound on the minimal size of a test set that provides edge covering was inferred by McCabe (1976) in an article describing the relationships between control graph structure and complexity. Because of the importance of this article in describing the graph relationships that hold for programs, a synopsis of the results follows.

A strongly connected (directed) graph is one in which any vertex can be reached from any other by some path through the graph. If an edge from the exit vertex (last program statement) to the entry vertex (first program statement) is added to a control graph G as in Figure 5, then G is strongly connected and a cycle (path from a vertex to itself) exists from every node. A definition from graph theory (Harary, 1972) specified that the cyclomatic number of a connected graph, V(G) equals e - n + 1 where e is the number of edges and n is the number of nodes. A resulting theorem (Berge, 1973) proved that V(G) is equal to the maximum number of linearly independent circuits in G. A set of linearly independent circuits of a graph can be combined to form any path in a graph, and contains no circuit that can be composed of any combination of the others. For any graph, there exist many such sets of linearly independent circuits.

As an example, consider the augmented control graph Figure 5 with six vertices and nine (program) edges. When augmented with edge 10, the result specifies that there exist 10 - 6 + 1 = 5 linearly independent circuits. Every path from a to f, when augmented with edge 10, forms a circuit. By definition, each path can be represented as a combination of any set of five linearly independent circuits,



One Set of Linearly

<u>Independent Circuits</u>

(a,b,e,f,a)

(b,e,b)

(a,b,e,a)

(a,c,f,a)

(a,d,c,f,a)

Figure 5. Control graph G with cycles

which form a basis for all circuits. Because the union of the edges composing the five linearly independent circuits exhausts the graph edge set, a maximum of five paths need to be used to cover the graph edges, even if each of the five paths employs a different linear independent circuit in its path description. Paige (1975) independently derived this bound on the maximum number of paths necessary to provide edge covering.

The cyclomatic number V(G) is only an upper bound on the minimum number of paths necessary to test every program branch. If one path can be composed of all V(G) independent cycles, then only one test data point is necessary from edge covering.

Other Structural Tests

Researchers and practitioners generally employ branch testing as a minimum measure of coverage (Brown & Lipow, 1975; Osterwell & Fosdick, 1976; Miller, 1976). It is often the case, however, that a larger set of test data is employed to exercise critical or interesting paths or to exercise particular segments under a variety of conditions to assess a program's reliability. For example, one test coverage criterion in Mills (1971) and Ramamoorthy, Kim, and Chen (1975) is called structured testing and has been shown to compare favorably to other test data selection criteria (Howden, 1978). As an approximation to full path-testing, structured testing dictates that all paths be tested that require fewer than an arbitrary number of iterations of any cycle. Variations in this strategy are often employed in cases where applying this criterion would leave part of a module untested because of complicated interdependencies between an outer loop index and inner loop bounds.

Another popular testing strategy requires test data that reside at boundary points at the input values. Other special values may be submitted to ensure that

- related data have distinct values
- certain arithmetic expressions involve zero-valued arguments
- nonnumeric inputs are submitted for each of the significant values that they may assume.

Test Data Selection

The problem of test data selection for structural testing is by no means solved upon the identification of the program paths to be tested. In fact, the problem of generating test data to execute any specified statement in a program is formally unsolvable. If a means can be found to ascertain a finite bound on program cycles (arbitrary bounds on loops), then a more reasonable problem results. The test data selection problem can be viewed as the identification of a data point x that invokes the execution of the path p(x) consisting of a finite number of functional statements sequenced by implicit (next-statement) and explicit (IF-THEN-ELSE, DO-WHILE) control structures.

A path is represented in a control graph by a sequence of edges. Recall that these edges may be labeled with conditions that serve as branch selectors when more than one edge emanates from a vertex. These conditions parallel the conditions found in corresponding program statements as the IF-THEN-ELSE, DO-WHILE, and CASE statement.

The input domain I associated with path p can be defined by the conjunction of these conditions. This conjunction represents a collection of predicates for which at least one desired test data point must serve as a solution. In general, finding this point is also an undecidable problem, even if the conditions are merely inequalities. Moreover, the conjoining of conditions into a path predicate is nontrivial, as it requires considering the effects of changes in the predicate arguments (free variables) made by assignment and other types of statements within segments.

The problem of path selection (by segment-covering, branch-covering, or other method) cannot in most instances be divorced from the determination of input data generating a path. This is due to the fact that abstract path selection based upon covering ignores the conditions that must hold for a path to be exercised. Many programs have a large number of <u>infeasible paths</u> whose path predicates evaluate to <u>false</u>. Howden (1978b) believed the presence of infeasible paths to be the most serious problem in path selection. Moranda (1978), in randomly exercising a numerical algorithm, found far fewer paths than would be expected if each path were feasible and equiprobable. Many of the designers of test generation systems ignore the possibility of infeasible paths, preferring to iterate between path selection and infeasibility determination until a covering is found (Miller, 1976). Many articles on test data generation merely ignore the problem (Hoffman, 1976; Miller & Melton, 1975).

Path infeasibility determination is a natural consequence of path predicate determination in test data generators. Test data generators can be roughly categorized by

- the degree to which they aid in path selection, and
- the degree to which they aid in path predicate evaluation.

RSVP, one of the earliest such systems, produced a branch-covering on FORTRAN programs and partially evaluated the path predicate by folding constant expressions and dropping obviously redundant path constraints (Miller & Melton, 1975). The resulting series of inequalities (virtually all FORTRAN predicates involve comparisons of numerical data) is printed out for the user to solve. PET, a Program Evaluation Tool developed by Stucki (1977), instruments a FORTRAN program with probes that indicate segment and branch usage. A testing system at TRW Systems, Inc., aids in path selection and attempts to evaluate path predicates composed of equalities by algebraic methods (Hoffman, 1972). Hence it appears that the computing community is a number of years away from developing systems that generate sets of nonredundant test data that cover all feasible paths.

Symbolic Execution

ATTEST, a system codeveloped by Clarke (1978) exhibits the most sophisticated approach to path selection and feasibility determination. A user may select from a set of testing criteria, including segment-covering, branch-covering, structured testing, and full path selection. ATTEST employs a linear programming algorithm to determine the minimum and maximum values for each loop in the path. The traversal of the control graph that searches for paths is heuristically driven and integrated with a "solver" routine that attempts to evaluate path predicates as they are composed. The solver employs linear programming methods on linear predicates and heuristic methods for more complex conditions.

In operation ATTEST closely resembles many aspects of a symbolic execution system. A symbolic execution system is usually initiated at the root (first executable statement) of a program. A set of input variables of indeterminate value is identified (these are subroutine parameters, COMMON variables, and the subjects of input statements). Every other program variable value is maintained as a function of the input variables. Hence every decision point's condition can be reexpressed in terms of input variables. During an interactive session with a symbolic execution system, a user is asked to specify which branch to take at each decision point. Each decision branch adds one more decision condition to a set that, when conjoined, specifies the path predicate, hence the input conditions that generate a path to the particular state at which the symbolic execution is stopped.

In EFFIGY, a symbolic execution system developed by King (1976), the user may save state information so that he may back up to an earlier decision point to take a different path of execution. Similar systems are under development elsewhere (Howden, 1977; Bayer, Elspas, & Levitt, 1975).

Data Flow Analysis

Although it falls outside the definition of testing, data flow analysis is a powerful validation method often integrated within symbolic execution systems. Data flow analyses involve the determination of when program variables are referenced and defined within an execution sequence. Data flow analysis can be performed during path enumeration or symbolic execution by determining, for each analyzed execution sequence, whether variables are used prior to their definition (assignment) or assigned a value and then not used. The identification of these and other data flow anomalies is the function of data flow analysis systems. Both the DAVE and ATTEST systems referenced above incorporate data flow analyses as an option during path generation.

Software Error Classes

A number of researchers have attempted to categorize program error types. The most extensive study of this kind was conducted by TRW Systems, Inc. (Thayer, Lipow, & Nelson, 1976), and included an analysis of error reports collected during the development of four large software systems. A great number of automated analysis tools was brought to bear on the systems' source code and on the error-recording documentation generated during the course of systems development. A very extensive list of error classes and subclasses resulted, including

- Computational errors, resulting from
  --the improper coding of formulae used in problem solving (e.g., quadratic roots equation)
  --bookkeeping (e.g., array index or record number calculation)
- Logic errors, in which
  --missing, excess, or erroneous conditions were discovered
  --functional sequencing was incorrect for some case
  --an algorithm realized a function other than what was desired

--physical characteristics of the problem were misunderstood (e.g., memory requirements, time duration, device interface)

- Input/output errors

  --improper format or timing of output

  --excessive or insufficient results display

- Data handling errors

  --data structures undefined or uninitialized

  --improper data types used for an operation

  --attempts to access data outside of legitimate address space or data structure bounds

- Interface errors, in which global, shared, or passed data were incompatible or improperly referenced from routine to routine, routine to database, or system to external environment

- Standards violations, in which nonconformance of program text to coding standards was discovered or incomplete documentation was provided.

Because these "errors" were recorded under all conditions of improper system performance or representation, some of the designated errors, in fact, describe "failures," including failures to meet requirements and failures resulting from operator error. The sources of these errors were attributed to development phases (requirements, design, coding, maintenance) depending upon the time when error-producing decisions were first made (Thayer et al., 1976). The frequency of the major error categories for three of the four projects is shown in Table 2. The category "Other" was used to group errors resulting from improper system analysis or design.

Table 2

Empirical Frequencies of Major Error Categories

Error Category	Number of Errors	Percentage of Code Based Errors	Percentage of Total Errors
Computational	552	12	9
Logic	1,333	29	20
Input/output	911	20	14
Data handling	887	19	14
Interface	932	20	14
Total code based	4,615	100	71
Standards	391		6
Other	1,470		23
Total	6,476		100

Note. From Software reliability study by Thayer, Lipow, & Nelson. Redondo Beach, Calif.: TRW, March 1976, p.

Rubey, Dana, and Biche (1975) analyzed a collection of data obtained from a variety of (undisclosed) medium-scale assembly language development efforts. In developing an empirically-based estimating formula for validation cost, the authors reviewed the type and frequencies of errors discovered during the validation phase. The error classes defined by these researchers coincide approximately with those employed in the TRW study as shown in Table 3.

Table 3

Empirical Frequencies of Major Error Categories

Error Category	Number of Errors	Percentage of Code Based Errors	Percentage of Total Errors
Computational	113	30	9
Wrong operation (69)			
Poor scaling (22)			
Other (22)			
Data access	120	32	10
Logic/sequencing	139	37	12
Wrong condition (28)			
Sequencing (98)			
Other (13)			
Total code based	372	100	31
Specification	485		40
Standards violation	118		10
Documentation	96		8
Total	1,202		100

Note. From Quantitative aspects of software validation by Rubey, Dana, & Biche, IEEE Transactions on Software Engineering, June 1975, SE-1(2),

One significant finding of this study was the apparent usefulness of static analysis (e.g., data flow analyses, cross reference checks, code review). The application of static analysis techniques resulted in the discovery of about half of the errors found, and these errors were discovered early in the validation effort at less cost than execution-based testing. Alberts (1976) cited a similar efficacy for static analysis, reporting that 46 percent of logic and coding errors were detectable by manual inspection and formal static methods.

In a recent paper, Fujii (1977) outlined the validation activities performable in each phase of system development. Her classification of implementation errors includes logical/branching, data accessing, and sequencing, defined in a manner very similar to the studies mentioned above. Fujii's analysis refers to medium to large programs with high reliability requirements developed under contract to the Department of Defense.

Very few data have been published on error frequencies in small software. Because small projects seldom have the budgetary luxury of gathering development statistics, one can only infer from intuition and other researchers' comments that the error composition for small software is most heavily concentrated in physical design and implementation flaws, with a higher frequency of computational, logic, and sequencing errors. Over a period of years, Howden (1978) studied the efficacy of various testing procedures on small programs written in higher-level languages. Each of the following testing approaches was employed and the number of errors discovered was tallied for each:

Testing Approach	Number of Errors Discovered
Path testing	18
Branch covering	6
Structured testing	12
Special values testing	17
Interface testing	2
Anomaly detection	4
Specification-based testing	7
Total errors	28

The relative effectiveness of path-based and special values (boundary points) techniques and lesser efficacy of formal specification, interface, and anomaly-detection approaches supports the contention that errors in requirements and interface definition are less prevalent in the small software development domain. The high proportion of "simple" implementation errors in small systems may justify optimism that small projects may be more mechanically tested at a reduced cost. Computation errors can typically be found by segment-covering approaches, because any path incorporating such an error may lead to failure. Logic errors are amenable to discovery through branch and special values testing, whereas sequencing errors may often be found by structured and other limited path testing. Each of the testing approaches can be facilitated by automated tools providing program instrumentation, test data selection, and failure detection (Ramamoorthy & Ho, 1975).

Although testing serves to purge programs of errors, it also operationalizes the assessment of reliability. Error discovery has always been a phenomenon greeted with mixed emotions by developers. Although the reliability of the code has undoubtedly been enhanced by error removal, one's confidence in program correctness may also suffer, depending upon the timing, frequency, and circumstances associated with the failure.

Failure circumstances are primarily dichotomized into the differences between testing and operational environments. Failures are provoked in testing, the object of a formal search for unfaithful translation of specifications. Failures in operations are avoided as much as possible, however, and any model of fault occurrence must distinguish between these program execution environments.

The difference between testing and development environments is often approached by considering the representativeness of the test data set applied to a software package. Littlewood (1979) stated that reliability models have not yet fully dealt with the question of representativeness. Thayer et al. (1976) attempted to address directly test representativeness by modifying a data-domain reliability estimator by an hypothesized operational profile. Brown and Lipow (1975) proposed the use of the X^2 statistic to measure the conformance of structural test sets with an assumed input distribution, and showed how additional test data points can help converge the testing and operational input profile. Testing theorists appear to ignore the issue altogether, apparently feeling that guarantees of testing thoroughness preclude the need to consider the distribution of program use (Howden, 1978; Ramamoorthy & Ho, 1975).

The differences in approaches hinge upon the theorist's preference for prediction and fear of failure. Testing costs are directly related to testing duration; testing duration is, in turn, dependent upon the growth of confidence in a program and can be enhanced by predictors of residual error count, time-to-next-failure, and failure rate. In a sense, testing proceeds until the cost of running without failure becomes prohibitive.

The situation in the operational environment is exactly opposite. Whereas the deferral of error correction to the operational phase may lead to increased debugging costs, it is the cost of failure that motivates formal testing. Littlewood (1979) proposed that we are currently in the second stage of reliability model development, the fourth of which will explicitly include operational failure costs in the lifecycle costs that affect development decisions. He, as well as other researchers, appreciates the problems of failure cost prediction. Unlike errors, whose cost of removal depends upon their subtlety and complexity, the cost of failure depends greatly upon the circumstances of program use. In real development environments the possibility of costly failures is considered and often motivates the systematic testing of infrequently used program functions. Viewed in this context, thorough nonrepresentative testing becomes a reasonable approach to risk aversion, even though the resulting reliability estimators may poorly predict testing duration and/or operational failure rate.

Testing Cost and Efficacy

The Testing Problem

Software developers are frequently faced with the problem of determining how much testing should be performed on a software good. One may attempt to end up with the same cost distribution per development phase as that recommended in the literature (Zelkowitz, 1978; Boehm, 1973), but these percentages vary a great deal and are usually applicable only to large, well-organized development teams working on large, critical systems. Most developers still determine a testing budget by informal estimation based upon experience with similar systems. As testing tools and strategies become more abundant, test planning becomes more complex, for the cost and efficacy of alternative approaches must be considered.

A variety of phase-dependent techniques has been discussed in previous sections. Although early validation of requirements and design is an important and worthwhile activity, this section focuses on the decision making in the code-testing phase. Code testing is a development step that no software developer can ignore, and so an analysis of the alternatives available in this phase is expected to have the most general applicability.

Code testing guidelines are particularly useful to the developers of small- to medium-scale software goods. The requirements for small software are usually well-defined and the moderate budgets allocated for these projects often preclude the use of formalisms in organization and representation that admit of modern validation techniques. Testing strategies and techniques, on the other hand, are better understood, generally more intuitive, and span a wide range of sophistication that permits the developer to choose a degree of formality suitable to the importance of the system and its reliability requirements.

Testing costs can be categorized by considering the steps involved in code validation. The transition from design to program code results from decisions including choice of programming language, algorithms and data structures, and target machine. Program validation procedures commence with examination of the coded program as represented in a human (coding sheet) or machine-readable form (cards, disk file). Syntactic validation is necessarily performed by a translator (compiler, interpreter) invoked to reduce the program text to an executable form. A translator reports on syntactic correctness by determining the program's conformance to the grammar associated with the programming language. Although sophisticated translators can perform many types of static analyses to assess semantic correctness, it is assumed here that separate tools are required to perform data flow analysis, reachability analysis, and other static validation procedures.

Sampling Strategies

When a program is free of syntactic errors, it remains to be tested for operational conformance to its desired function. Testing involves the application of data and this application involves the selection of representative inputs. This choice may be termed a sampling strategy because it requires the selection of one or more data points from the (generally) large set of possible program inputs. In the discussion in previous sections, the set of possible data inputs has been designated by the set D. Each element of D is a collection of inputs that forces a program P to proceed from an initial state to one or more states that signify completion. A sequence of program actions from initiation to termination is considered an execution or run and serves as the basis of analysis for correctness.

In some programs or systems this unit of execution may be difficult to assess: real-time code may continuously monitor and react to environmental stimuli; query systems may interact continuously with a variety of users; operating systems may continuously provide system services while controlling access to system resources. In nearly all cases, however, a run or complete execution can be defined. Program terminations serve as obvious boundaries for simple programs, whereas runs in continuous systems may be defined as execution sequences between transaction submittals or service requests.

Discovering a relevant definition for program inputs may also prove elusive. One may, however, simply consider the time-ordered data values submitted to a program during a run and designate each sequence of input data values $d = (d^1, d^2, \cdots)$ as an element of the input domain, D. In simple systems the ordering of the d^k's is unimportant, where, for example, each d^k represents a separate value and the entire set is submitted simultaneously at program start. A program may, however, acquire inputs during the course of execution and sequences of d^k's may be highly interdependent.

The cardinality of D is dependent upon the data types of the d^k's and the relationships among them. In simple programs each constituent d^k of an input data point d may be a separate variable of a common data type (say, integer or character). The cardinality of D, then, is at most the product of the cardinalities of each associated data type, where data type indicates a set of permissible variable values. This maximum cardinality is often bounded by interrelationships among program inputs. For example, in matrix multiplication the input matrices must be conformable (L by M and M by N) to be proper inputs.

Other bounds may be placed upon individual input values as well as permissible combinations. A programming language may provide only general data types (for example, integer) when an input value is specified to reside in a restricted range (say, gross income in a tax computation routine). One must recognize the collection of constraints on input values as this collection is important in determining the permissible execution paths through the program.

To reiterate, a sampling strategy requires the selection of sample points for application to a program P to attempt computation of the program function f. Informal sampling strategies abound, including pseudorandom, functional, descending-criticality, and most-obvious-first. A testing procedure is considered pseudorandom if the selection of test data is performed in a manner in which no methodology is apparent, and for which it appears that input sets are selected from D with equal probabilities. True random sampling may be performed, of course, by employing a mechanism to choose from D nondeterministically with the resulting sample $\tilde{d}$ distributed according to the mechanism's selection function.

Functional testing results from the application of test sets to determine the conformance of the program to its specifications. When the specifications imply some easily recognizable classes of desired program behavior, functional sampling attempts to confirm the correctness of the program functions associated with the desired behavior. A partitioning of the input domain is usually suggested by the set of desired functions (say, different transaction types in a bank's teller system) and functional testing requires a selection of data points within each partition. The selection within partitions may be performed by any other sampling technique.

Sampling in order of decreasing criticality refers to the informal notion of testing those program functions whose failure would preclude further testing, or which, if found incorrect, could prove costly. As an example of the former case, a frequently used subroutine may be validated first to allow a confident assessment of error types in other program parts that employ that routine (this is the motivation for bottom-up testing). The latter case refers to a test ordering in which a preference function is imposed upon the kinds of errors allowable in the program. One may assess the need for extensive testing of life support software for a space capsule, but commit fewer resources to testing less critical functions (e.g., caloric intake monitoring).

Closely associated with sampling by decreasing criticality is the modeling of sampling by removal of the _most obvious errors first_. In practice, a large number of errors is discovered in initial testing phases, with error discovery decreasing as time goes on. This phenomenon results because some error types are more obvious than others: for example, those involving a single statement rather than combinations of segments. Sets of related errors made uniformly throughout the program (e.g., improper declarations or subroutine calls) also tend to be discovered early and, depending upon the error discovery model, may be counted as a large number of single errors. Time-domain models that employ a decreasing hazard function can account for this phenomenon (see Chapter III).

Formal sampling strategies require a stratification of the data domain. _Structural testing_ involves a stratification with partitioning induced by the structure of the program. Structural testing is composed of three steps: structure determination, partitioning, and test data selection. _Structure determination_ involves the analysis of possible program execution paths and is normally performed by constructing a graph that indicates program segments and precedence relationships among them. Partitioning involves the recognition of disjoint subsets of D, each of which is associated with some aspect of program structure. Segment covering is one such sampling criterion in which the desired test data set must exercise every program statement. Branch covering requires that a set of data invoke the transfer between every connected segment pair. Structured testing requires the execution of every basic path.

The usual outcome of partitioning is a specification of the set of paths that satisfy a covering criterion. With each path is associated a path predicate characterizing the property of the domain subset whose members force the path's execution. To operationalize these partitionings, one or more data points must be defined for each subset. As discussed above, this determination may be nontrivial, requiring the solution of the path predicate in terms of the input variables.

Sampling Cost and Effectiveness

The costs and benefits of test data selection arise in two general ways: sampling cost and sampling efficacy. Sampling cost relates to the time, effort, and money expended in constructing a test data set conforming to specified testing criteria. For random, pseudorandom, or functional testing this cost may be minimal because these methods are relatively intuitive. Moreover, inexpensive test tools exist to aid in these simple forms of test data construction (Reefer &

Trattner, 1977). The cost of sampling subject to structural testing constraints is greater because it involves

- the construction of a program graph
- the selection of paths providing the desired test coverage
- the determination of path feasibility by definition of data points forcing each path's execution.

The cost of graph construction if basically linear in the number of statements, although manual construction requires a slightly higher order of effort for large and cumbersome graphs (Aho & Ullmann, 1975). Path selection time is also proportional to the number of segments or branches for their respective coverings, and often is immediately obvious in manual analysis of small graphs (Gabow, Maheshwari, & Osterweil, 1976). Basic path selection is also linear in the number of branches, although more difficult to perform manually (Paige, 1975). More complex criteria may require execution time that is a polynomial function of the segments or branches. Full path testing is generally impractical for all but the smallest of programs.

Feasibility analysis and data point selection are formally unsolvable, but some current systems do perform this function for path predicates of special forms. Path predicate composition is linear in the number of statements on the path. Path predicate solution depends upon the efficacy of the routine employed for feasibility analysis and depends upon (at least) the number and type of path conditions (constraints).

Sampling effectiveness is related to the "speed" with which a sampling technique discovers errors. The application of <u>different</u> test points is clearly more efficient than that of simple random test points, because duplication is precluded in the former. In a number of studies, Howden (1976, 1978) compared the observed relative efficacies of various test criteria (segment covering, branch covering, structured testing, full path testing) but failed to give any measure of the variability of each criterion's error discovery.

The efficacy of a particular data set selection approach should be reflected in the reliability model employed in assessing program correctness. More effective sampling strategies should increase the detection rate in time-domain models as reflected in the hazard function. Data-domain models should show faster reliability growth and/or faster reliability estimate convergence.

Cost of Test Data Application and Fault Detection

Associated with each test data point or set are the costs of its application and results-assessment. A test run may be a simple procedure in which initial data are input interactively or a complicated procedure in which test drivers are required to simulate environmental inputs. In either case, it is probably reasonable to assume that each test case incurs a fixed cost of application, although this cost may vary greatly from system to system.

To be tested, a data point must invoke program execution, which requires computing resources. Many programs require computing resources (CPU time, memory) in quantities related to the values of input values. The time and space efficiency orders of tested programs dictate this resource use and hence impose a cost of test execution dependent upon the values of test data input. Test data values (d^k's) that affect data structure size (e.g., matrix order) or loop bounds (e.g., convergence tolerance) can vastly affect the cost of test data application. In practice, running times appear to be the more costly variable. Running time (elapsed or CPU) is directly related to the length of the execution path associated with each data point.

Fault detection involves the examination of test results to confirm correctness. The assessment of output correctness requires an independent evaluation of the program's function on the test input by what is termed a test oracle. Fault detection is easily performed for simple programs whose function is in some sense invertible (say, a square root routine), and can be performed inexpensively by a human oracle. Output from large complex systems is generally much more difficult to assess and often requires a simulation or execution of another similar program for output validation. It is probably reasonable to assume, however, that the cost of a test oracle is linear in the number of test data points. Where correctness can be determined only by comparison with a similar system the cost of detection may resemble that of test execution, but in many cases correctness of an output can be determined at a cost independent of the program time and space necessary for its computation.

Cost of Error Detection and Correction

Once a test input is found to cause premature program termination or generation of improper output, a search proceeds for the cause(s) of the fault. These causes are normally termed errors and their detection normally requires code review to determine the failure source(s). Error detection is still the most artful of programming practices, depending upon deep knowledge of the program, vagaries of the programming language employed, and other subtleties. It is probably less than realistic to assume that error detection costs are uniform over all program inputs. In the same way that prevalent failures occur early in testing, obvious errors are generally found first. In fact, part of the appeal of decreasing hazard function time-domain models is the way in which this phenomenon can be represented.

Error correction involves the rectification of program deficiencies by changing the text to reflect properly the functional specifications. Error discovery is closely associated with error correction and it is usually possible to integrate these costs as one function. Textual changes are normally followed by retranslation of the source to object program. Retranslation costs are usually polynomial in the number of statements and, in most cases, linear. Retranslation costs and turnaround times are often sufficiently great, however, that collections of test inputs are often batched so that a number of corrections can be made between test submittals.

The Cost of Errors in the Operational Phase

The costs associated with program errors that persist into the operational phase can be dichotomized into two general categories: the cost of failure and the cost of maintenance. The cost of failure is the prime determinant of reliability requirements and can include

- opportunity losses, organizational disruption, and other effects of system service suspension
- financial loss through improper accounting, over- or under-utilization of resources, improper purchases or sales, or other real loss

  .

  .

  .

- jeopardizing of human life.

The consequences of program failure to the developer-user are the direct kinds enumerated above. The software supplier may incur many indirect costs when a product is found to be unreliable, including

- loss of sales as a result of product reputation for unreliability
- litigation and settlement costs incurred as the result of program misperformance.

Both users-developers and software suppliers generally incur the cost of maintenance. The developer may or may not have the responsibility to maintain customer versions of a product, but must acknowledge errors reported by users and correct them if continued sales of the product are desired. As an approximation to reality, one may assume that all of these costs can be integrated into a simple function of the number of failures. Each failure incurs a fixed cost representing the damage caused by the misperformance, and a maintenance cost corresponding to some multiple of the error detection and correction costs of testing. These costs are usually a function of the number of remaining errors, n_e, but not necessarily linear in n_e (for time-domain models other than exponential).

No one has yet proposed a model which would dictate the type and duration of testing depending upon the costs of sampling, application, error detection/correction and program failure outlined above. Most researchers would agree with Littlewood (1979) that it is premature to attempt such a model before the computing community has reached a consensus on the proper mix of tools and approaches for different software development situations. In the sections above, the cost order of magnitude for each testing for each testing activity and result has been inferred. In this study experimental results will be reported in hopes of laying the groundwork for an economic model of testing. We believe that such an economic model is within the construction capabilities of the Software Reliability community and we offer this study as a step toward its resolution.

CHAPTER III

SOFTWARE RELIABILITY THEORY AND PRACTICE

Introduction

Software reliability has been a topic of much study and many publications during the current decade. Software reliability relates to the ability of a software system to perform as expected and is often associated with the degree to which embedded errors can cause system failure or misperformance. In a recent paper, Schick and Wolverton (1978) reviewed competing models proposed to relate system reliability to program errors and failure distributions. These approaches are generally based upon differing models of the probability distributions underlying fault occurrence and error frequencies as functions of system structure and use.

All software reliability models recognize a relationship between the existence of embedded errors and the time-distributed failures that occur when such errors are "discovered" during program execution. Schick and Wolverton dichotomized modern approaches to software reliability assessment as either time-domain or data-domain models. Time-domain models emphasize the failure distribution by hypothesizing error discovery as a function of some measure of time. Differing measures of time include the execution duration of the program (CPU time), real (calendar) time intervals, and the number of separate applications of input to the program.

Data-domain models define program reliability statistically in terms of errors per textual unit. The emphasis in data domain modeling is the assessment of error content as a proportion of the program's statements, object instructions, or executable paths.

Time-Domain Models

Mathematical Preliminaries

All of the most referenced time-domain models define reliability $R(t)$ as the probability of fault-less program execution over time t, hence

$$R(t) = 1 - F(t)$$

where
$$F(t) = \int_0^t f(x)dx$$

and $f(t)$ defines the interarrival density (limit of the probability that the first error is discovered at time t hence). Following Schick and Wolverton (1978), define

$$z(t) = \frac{f(t)}{1-F(t)} = \frac{F'(t)}{R(t)} = -\frac{R'(t)}{R(t)}$$

as the instantaneous error rate. It is also termed a hazard function. Solving the differential equation

$$z(t) = -\frac{R'(t)}{R(t)}$$

yields
$$R(T) = e^{-\int_0^t z(x)dx} .$$

A variety of proposed models can be analyzed on the basis of the terms developed above. The assumption of constant error discovery rate ($z(t)=\alpha$) leads to a standard exponential distribution

$$f(t) = \alpha e^{-\alpha t} \qquad R(t) = e^{-\alpha t} \quad t > 0;\ \alpha > 0.$$

Assumptions of increasing or decreasing error rate (modeling error correction) can be expressed by

$$z(t) = \alpha\beta t^{\beta-1} \qquad t > 0;\ \alpha,\beta > 0$$

yielding a two-parameter Weibull distribution for the error inter-arrival density

$$f(t) = \alpha\beta t^{\beta-1} \cdot e^{-\alpha t \beta} \qquad t > 0;\ \alpha,\beta > 0$$

and reliability function

$$R(t) = e^{-\alpha t^{\beta}} \qquad t > 0;\ \alpha,\beta > 0\ .$$

Early Models of Finite Error Content

If error discovery rate is proportional to the number of remaining embedded errors, then a more representative model of time-varying reliability may be proposed. Assuming that the likelihood of detection of each of the n_e errors is independent, then the error discovery rate for the i^{th} error may be written as the constant function

$$z_i(t) = \alpha_i = \varphi(n_e - (i - 1))$$

where φ is the instantaneous discovery rate for one error. Jelinski and Moranda (1973) proposed this model designating

$$f_i(t_i) = \alpha_i e^{-\alpha_i t_i}$$

as the interarrival time density function for time intervals $t_1, t_2, \cdots, t_{n_e}$ between successive error discoveries.

Schick and Wolverton (1972) ventured the assumption that the longer the time period from the last error discovery, the greater the likelihood of discovery. A formalization of this notion is the error rate function written as

$$z_i(t) = \varphi(n_e - (i - 1))t$$

where φ is the original instantaneous discovery rate, t is the time since the last error discovery, and n_e and i are defined as above. The associated interarrival density and reliability functions are

$$f_i(t_i) = z_i(t_i)e^{-z_i(t_i)\frac{t_i}{2}}$$

$$R_i(t_i) = e^{-\varphi[n_e - (i - 1)]\frac{t_i^2}{2}} .$$

Both the Schick-Wolverton (1972) and Jelinski-Moranda (1973) models may be generalized realistically to admit the possibility that more than one error is found during testing intervals. Thayer et al. (1976) proposed extensions, as

$$\text{Jelinski-Moranda:} \quad z_i(t_i) = \alpha_i = \varphi(n_e - n_{i-1})$$

$$\text{Schick-Wolverton:} \quad z_i(t_i) = \varphi(n_e - n_{i-1})t_i$$

where n_i represents the cumulative number of errors found prior to the i^{th} testing interval and $n_0 = 0$.

As stated by Schick and Wolverton (1978), "the reliability analyst should expect his time-domain model to be a predictor for both the number of errors remaining and the mean time for the next error to occur" (p. 2). The time-domain models cited above can be employed in these regards a^ter some testing has taken place, if estimators can be used in place of unknown parameters. Thayer et al. (1976) derived or cited maximum likelihood estimators (MLEs) for the α's and n_e above, as well as developing asymptotic variances and correlation coefficients for these parameters. The general procedure for determining the MLEs involves the simultaneous solution of two equations in α and n_e (or φ and n_e). Approximate confidence regions for the parameters may then be constructed using the asymptotic variances.

Goel's Model

In a recent paper, Goel (1978) proposed a time-domain model in which failure interarrival times were considered the result of a non-homogeneous Poisson process (NHPP). Goel hypothesized a continuous function n(t) giving the cumulative number of software failures occurring by time t expended during validation. Because each failure is assumed to be the result of a unique error, and because each error is assumed to be corrected subsequent to failure, n(t) is a nondecreasing function bounded by n_e, the total number of errors in the program. By assuming that the number of errors detected in the interval $(t, t + \Delta t)$ is proportional to the number of undetected errors, Goel deduced that

$$n(t) = n_e(1 - e^{-\varphi t})$$

with φ the constant of proportionality.

Goel (1978) imposed the Poisson postulates upon the random variable $\tilde{n}(t)$, as

- $\tilde{n}(0) = 0$
- each $t_1, t_2, \cdots$ is statistically independent, where $t_i = \tau;\ \tilde{n}(\tau) = i$
- Pr (2 or more events in $(t, t + h)) = o(h)$
- Pr (exactly one event in $(t, t + h)) = \lambda(t)h + o(h)$

to derive a Poisson distribution for $n(t)$, as

$$f_{\tilde{n}(t)}(k) = \Pr(\tilde{n}(t) = k) = \frac{m(t)^k\, e^{-m(t)}}{k!}$$

for
$$m(t) = \int_0^t \lambda(s)\, ds \ .$$

The mass function $f_{\tilde{n}(t)}(k)$ has mean $m(t)$, which Goel chose to equate to the assumed deterministic function $n(t)$ as:

$$m(t) = n_e(1 - e^{-\varphi t}) \ .$$

To summarize the assumptions given above, the time distribution of failures, $n(t)$, is assumed to be a member of the family of functions $\{n(t;n_e,\varphi)\}$. In the absence of knowledge regarding the values for n_e and λ, a Poisson prior distribution for $\tilde{n}(t)$ is given as $f_{\tilde{n}(t)}(k)$ with mean

$$m(t) = \int_0^t \lambda(s)\, ds = n_e(1 - e^{-\varphi\tau})$$

to any given time t. For a given n_e and φ one obtains

$$\Pr(\tilde{n}(\infty) = k) = \frac{n_e^{k} \cdot e^{-n_e}}{k!}$$

a Poisson distribution for the number of failures over a debugging interval of indefinite length.

Letting $\tilde{\tilde{n}}(t)$ indicate the number of errors remaining at time t, one obtains mean and variance

$$E[\tilde{\tilde{n}}(t)] = E[\tilde{n}(\infty) - \tilde{n}(t)] = n_e \cdot e^{-\varphi t}$$

and $$\mathrm{Var}[\tilde{\tilde{n}}(t)] = n_e + n_e(1 - e^{-\varphi t}) - 2n_e(1 - e^{-\varphi t}) .$$

Moreover, if y is the number of errors found by time s, then the conditional distribution of $\tilde{\tilde{n}}(s)$, given this information, is

$$\Pr(\tilde{\tilde{n}}(s) = k | \tilde{n}(s) = y) = \Pr(\tilde{n}(\infty) = y + k)$$

$$= \frac{n_e^{(y+k)} \, e^{-n_e}}{(y + k!)}$$

with mean $$E[\tilde{\tilde{n}}(s) | \tilde{n}(s)=y] = n_e - y .$$

The reliability of the program over time t, starting at this time s, is given by

$$R(t) = e^{-n_e[e^{-\varphi s} - e^{-\varphi(s+t)}]} .$$

As a result of choosing a nonhomogeneous Poisson process to describe $\tilde{n}(t)$, Goel (1978) was able to derive maximum likelihood estimators for parameters n_e and φ. Given a sample of observed validation times and associated cumulative error counts, $T = \{(y_1,t_1),(y_2,t_2,\cdots,(y_k,t_k)\}$, these estimators can be calculated. The independence of

$\tilde{n}(t_i)$'s permits the joint mass functions, $\Pr(\tilde{n}(t_1)=y_1, \tilde{n}(t_2)=y_2, \cdots, \tilde{n}(t_k)=y_k)$ to be computed as the product of the marginal distributions $\Pr(\tilde{n}(t_i)=y_i; n_e, \varphi)$, yielding likelihood function $L(n_e, \varphi)$ with global maximum $L^*(\hat{n}_e, \hat{\varphi}; T)$ occurring where

$$\hat{n}_e(1 - e^{-\hat{\varphi}t_k}) = y_k$$

and
$$\hat{n}_e t_k e^{-\hat{\varphi}t_k} = \sum_{1 \le i \le k} \frac{(y_i - y_{i-1})\left(t_i e^{-\hat{\varphi}t_i} - t_{i-1} e^{-\hat{\varphi}t_{i-1}}\right)}{e^{-\hat{\varphi}t_i} - e^{-\hat{\varphi}t_{i-1}}} .$$

Goel provided numerical solutions for two sets of data that provide a reasonable fit to the empirical failure times. By invoking asymptotic normality for the maximum likelihood estimators, Goel computed a covariance matrix for $(\hat{n}_e, \hat{\varphi})$ and provided a time-varying estimate and confidence band for the actual number of errors, $\tilde{n}(\infty)$, and the number of remaining errors, $\tilde{\bar{n}}(t)$. The actual error predictor $\hat{n}(t)$ appears quite good (as it appears on the reported plots), whereas the remaining error-predictor, $\hat{\bar{n}}(t)$, provides a reasonable fit to the empirical results for all but large values of t. This underestimation of $\bar{n}(t)$ for large t would tend to justify Littlewood and Verrall's (1973) claim that exponential models of failure interarrival times are deficient near the end of the validation process.

A Bayesian Approach

A number of investigators have proposed reliability models in which the distributions of interest (failure interarrival, reliability, etc.) are updated to reflect the information gained during program testing (Littlewood & Verrall, 1973; Goel & Okumoto, 1978). In a recent work, Littlewood (1979) argued that a proper measure of program reliability is performance-orinted, and that the distribution of future failures calls for a subjectivist's view of this nondeterministic phenomenon.

Littlewood (1979) offered a model in which the source of uncertainty regarding failures is dichotomized between the nondeterminism of program inputs and the probability that an input leads to failure. Littlewood hypothesized an input set D and regarded the subset $\tilde{D}^e$ that leads to program failure as random, because different software development efforts lead to different partitionings of D into correct and failure-producing input subsets, $\tilde{D}^e$ and $\tilde{D}^c$.

As a program is debugged $\tilde{D}^e$ changes, as denoted by the sequence

$$\tilde{D}_1^e,\ \tilde{D}_2^e,\ \ldots,\ \tilde{D}_j^e,\ \ldots$$

corresponding to the failure-producing input sets associated with each program version. Associated with each D_j^e is a failure rate, λ_j, which serves as a parameter in the time-to-next-failure distribution

$$f(t_j|\lambda_j) = \lambda_j\, e^{-\lambda_j t_j}\ .$$

Littlewood (1979) offered the gamma distribution as an appropriate family for modeling uncertainty regarding λ_j,

$$f_\Gamma(\lambda_j) = \frac{\beta_j\ \lambda_j^{\alpha-1}\ e^{-\beta_j\lambda_j}}{\Gamma(\alpha)}\ .$$

The mixing of these two distributions yields a member of the Pareto family of distributions:

$$f(t_j|\alpha,\beta_j) = {}_0\int^{\infty} f(t_j|\lambda_j) \cdot f(\lambda_j|\alpha,\beta_j)\, d\lambda_j$$

$$= \frac{\alpha\beta_j^{\ \alpha}}{[t_j + \beta_j]^{\alpha+1}} \cdot$$

The parameter β_j serves as a "growth function" of reliability over discrete time measured by j, and Littlewood (1979), unlike other researchers, suggested that this reliability growth is linear neither in time nor in the number of discovered errors. Rather, he suggested that most serious errors are detected early. This phenomenon can be modeled in the following way. Consider the program with N initial bugs, subjected to testing in which error correction (bug removal) occurs with certainty. Assuming that the independent failure rate ν_i of each bug is identically distributed with mean ν, then the failure rate between successive error discoveries is given by

$$\lambda_j = \nu_j + \nu_{j+1} + \cdots + \nu_N \,.$$

Letting

$$\Delta_j = \sum_{1\leq i\leq j} t_i$$

denote the cumulative time to the j^{th} failure, then one may express the conditional pdf of each undiscovered bug's error rate, ν_j, as

$$f(\nu \mid \text{bug undiscovered in internal } (0,\Delta_{j-1}))$$

$$= \frac{\Pr(\text{bug not eliminated in time interval } (0,\Delta_{j-1}) \mid \nu)\, f(\nu)}{\int_0^\infty \Pr(\text{bug not eliminated in time interval } (0,\Delta_{j-1}) \mid \nu) f(\nu) d\nu}$$

$$= \frac{f(\nu)\; e^{(-\nu\Delta_{j-1})}}{\int_0^\infty f(\nu)\; e^{(-\nu\Delta_{j-1})}\, d\nu} \;.$$

If a single error's failure rate is modeled as distributed according to

$$f_\Gamma(\nu;\varphi,\beta) = \frac{\beta^\varphi \nu^{\varphi-1}\, e^{-\beta\nu}}{\Gamma(\varphi)}$$

then the conditional distribution $f(\nu \mid \text{bug undiscovered in } [0,\Delta_{j-1}])$ becomes

$$f_\Gamma(\varphi,\beta + \Delta_j)$$

and the distribution of λ_j becomes

$$f_\Gamma((N- j + 1)\cdot\varphi,\beta + \Delta_{j-1})$$

with mean

$$\frac{(N - j + 1)\varphi}{\beta + \Delta_{j-1}}$$

serving as the hazard function $z(\Delta_j)$. Given the foregoing assumptions, Littlewood (1979) concluded with the unconditional failure interarrival distribution

$$f(t_j;\varphi,\beta) = \int_0^{\infty} f(t_j;\lambda_j)\, f(\lambda_j;\varphi,\beta,\Delta_{j-1})\, d\lambda_j$$

$$= \frac{(N - j + 1)\varphi(\beta + \Delta_{j-1})^{\varphi(N - j + 1)}}{(\beta + \Delta_j)^{(N - j + 1)(\varphi + 1)}} .$$

The first model presented above with

$$f(t_j|\alpha,\beta) = \int_0^{\infty} f(t_j|\lambda_j) \cdot f(\lambda_j|\alpha,B_j)\, d\lambda_j = \frac{\alpha\beta_j^{\alpha}}{[t_j + \beta_j]^{\alpha+1}}$$

is termed the Littlewood-Verrall (L/V) model, as it was first presented by these researchers in an early joint paper (1973). Littlewood (1979) applied this model to a set of failure interarrival time data first presented and analyzed by Musa (1975). The author assumed a linear reliability growth function,

$$\beta_j = a_0 + a_1 \cdot j$$

and obtained the maximum likelihood estimators $\hat{a}_0$, $\hat{a}_1$, and $\hat{\alpha}$ over the first few observations $t_1, t_2, \cdots, t_k$ and employed $f(t_j|\hat{\alpha},\hat{\beta}_j=\hat{a}_0 + \hat{a}_i \cdot j)$ to derive predictions of future failure rate, mean time to failure, and reliability. Littlewood (1979) obtained an excellent fit of predicted to actual times $t_{k+1}, \cdots, t_n$ using this model.

A Model Based upon Order Statistics

There is every reason to believe that the order in which errors are found is nondeterministic; current software reliability theory models, however, uniformly model the failure phenomenon as successive events, each with a related problem distribution. At face value, it would seem an unjustified research decision to ignore the probability of underlying events (errors "waiting" in parallel time to be discovered) and jump directly to models of the distributions of ordered events. That no one questions this common research practice (see Goel, 1978: Schick & Wolverton, 1978; Littlewood, 1979) is understandable, for the assumptions of decreasing failure rate and negative exponential interarrival times is in conformance with assumptions of

- equal failure rates for each constituent error, and
- negative exponential failure time distributions for each error.

In short, previous investigators have been able to ignore the order in which failures occur by modeling the associated errors as homogeneous Poisson processes. If, however, either of the assumptions above is invalid, no reasonable model of failure times can be considered without consideration of the failure behavior of each individual error.

The assumption that underlying errors are not identically distributed allows one to state simpler bases for a failure model:

- If errors are assigned arbitrary error rates $\lambda_1, \cdots, \lambda_n$, and hypothesized to fail under Poisson assumptions in parallel time, then no general or common models of ordered failure time distributions are possible.
- Unless some relationship is assumed among the probability distributions of underlying errors, no inference can be made about unobserved errors.
- It is counter-intuitive and not in keeping with most programmers' experiences that all errors are equally "hard," or randomly pursued.

A model is proposed that is based only upon these simpler assumptions of error independence and individuality. The specific assumptions are:

- Each error that is inadvertently introduced into a program can be assigned a "difficulty rating" λ_i; this difficulty rating reflects the combination of its inherent subtleness or (conversely) ease of discovery, and the relative skill or predisposition of the tester in pursuit of this kind of error.
- These difficulty ratings can be considered random variables, identically distributed as $f_{\tilde{\lambda}}(\lambda_i)$, reflecting the imperfect program development skills of every software practitioner.
- Software failures (either improper program termination or production of incorrect results) are induced by the submission of test data and identified by code and results review; this procedure can be modeled as a sampling procedure, and the time $\tilde{t}_i$ to discovery of an arbitrary error, ϵ_i, is a random variable; each unordered failure time is subject to the same family of distributions $f_{\tilde{t}}(t;\lambda_i)$, which differ only in the value of their sole parameter λ_i.
- The resulting distribution of the failure time for the ith error is a mixture of the form

  $$f_{\tilde{t}_i}(x) = f_{\tilde{t}}(x) = \int f(x;\lambda_i)\ f(\lambda_i)d\lambda_i$$

  an identical distribution for each error, with cumulative distribution function $F_t(t)$.
- Because the most useful indexing of failures is by their order of occurrence, the random variable of interest is $\tilde{t}_{(i)}$, the ith order statistic "drawn" from a population of n program errors, with distribution

  $$f_{\tilde{t}_{(i)}}(x) = i\binom{n}{i}f_{\tilde{t}}(x)\ F_{\tilde{t}}(x)^{i-1}[1 - F_{\tilde{t}}(x)]^{n-i}$$

as prescribed by results from the theory or order statistics (David, 1970).

The remainder of this section shows how these results are used in the development of a new time-domain model.

One may assume that a programmer makes n errors, $\{\epsilon_1, \epsilon_2, \cdots, \epsilon_n\}$ over the course of software development and that these are identifiable and correctable within the body of the program. With each error ϵ_i is associated a difficulty index λ_i that serves to represent the degree of effort, insight, and luck necessary to provoke and/or find this error. Because these errors are unintentional, one may assume that their introduction is stochastic and, further, that the continuum of "hard" to "easy" errors is reflected in the probability distribution for λ, of which each λ_i is a particular one. One may, for example, assume that the set of $\{\lambda_1, \lambda_2, \cdots, \lambda_n\}$ constitutes a sample from a gamma distributed "difficulty index generator" where

$$f_{\tilde{\lambda}}(\lambda_i;\alpha,\beta) = \frac{\alpha^{\beta} \lambda_i^{\beta-1} e^{-\alpha\lambda_i}}{\Gamma(\beta)} .$$

It is normal to assume that debugging activity is nonspecific--that is, that each error is the object of a simultaneous search for all errors (Musa, 1975). If this is the case, then each failure time $\tilde{t}_i$ (time to evidencing of the i^{th} error) is equivalently distributed and most likely parameterized by its difficulty index λ_i. If λ_i is interpreted as an arrival rate, and the Poisson postulates assumed for $\tilde{t}_i$, then the negative exponential distribution describes the conditional distribution of $\tilde{t}_i$ on λ_i:

$$f(t_i|\lambda_i) = \lambda_i e^{-\lambda_i t_i} .$$

Each t_i is an instance of a random process that generates failure times; this process has an unconditional distribution of

$$f(t) = f(t|\lambda)f(\lambda)d\lambda$$

which for the prior assumed distribution is a gamma mixture of exponentials of the form

$$f(t|\alpha,\beta) = {}_0\int^{\infty} f(t|\lambda)f(\lambda)d\lambda = {}_0\int^{\infty} \frac{\lambda e^{-\lambda t}\, \alpha^{\beta}\lambda^{(\beta-1)}\, e^{-\alpha\lambda} d\lambda}{\Gamma(\beta)}$$

$$= \frac{\alpha^{\beta}}{\Gamma(\beta)}\, {}_0\int^{\infty} \lambda^{\beta} e^{-\lambda(t+\alpha)} d\lambda = \frac{\alpha^{\beta}}{\Gamma(\beta)} \cdot \frac{\Gamma(\beta+1)}{(t+\alpha)^{\beta+1}}$$

$$= \frac{\alpha^{\beta}\beta}{(t+\alpha)^{\beta+1}}$$

which will be denoted as $f_{GE}(t|\alpha,\beta)$.

As a program is debugged, the portion of unscrutinized code and the subset of untested input domain shrink. Hence it may be more reasonable to assume that the instantaneous "failure rate" over time is

$$z(t) = \lambda t$$

as is assumed in the Schick-Wolverton (1972) model. Then the distribution of failure times should follow a Rayleigh distribution:

$$f(t|\lambda) = (\lambda t)e^{-\lambda t \cdot t/z} = \lambda t e^{-\lambda t^2/2} .$$

The resulting unconditional distribution of t is a gamma mixture of

Rayleigh distributed random variables of the form:

$$f(t|\alpha,\beta = {}_0\int^{\infty} \frac{\lambda t e^{-\lambda t^2/2} \cdot \alpha^{\beta} \cdot \lambda^{\beta-1}\; e^{-\alpha\lambda} d\lambda}{\Gamma(\beta)} = \frac{t\alpha^{\beta}B}{(t^2+\alpha)^{\beta+1}}$$

which will be termed $f_{GR}(t|\alpha,\beta)$.

The arrival times of errors may be indexed in order of their occurrence, as

$$\tilde{t}_{(1)} \leq \tilde{t}_{(2)} \leq \cdots, \leq \tilde{t}_{(n)} \; .$$

Because each of the underlying failure times $\tilde{t}_1, \tilde{t}_2, \cdots, \tilde{t}_n$ is identically distributed, one may draw on the results of the theory of order statistics. Immediately useful results include:

$$f_{\tilde{t}_{(i)}}(x;n) = i\binom{n}{i} F_{\tilde{t}}(x)^{i-1}[1 - F_{\tilde{t}}(x)]^{n-i} \; f_{\tilde{t}}(x)$$

$$f_{\tilde{t}_{(1)}, \cdots, \tilde{t}_{(n)}}(x_{(1)}, \cdots, x_{(n)}) =$$

$$\frac{n!}{(n-k)!} \prod_{j=1}^{k} f_{\tilde{t}}(x_{(j)})[1 - F_{\tilde{t}}(x_{(k)})]^{n-k}$$

where $f_{\tilde{t}}$ and $F_{\tilde{t}}$ are the density and cumulative distribution functions for the underlying distribution, respectively.

If each error committed in program development is equally "difficult," and $\lambda_i = \lambda$ for all i, then each failure time $\tilde{t}_i$ is identically distributed as

$$f_{\tilde{t}}(t;\lambda) = \lambda e^{-\lambda t}$$

and

$$f_{\tilde{t}_{(1)}, \ldots, \tilde{t}_{(k)}}(x_{(1)}, \ldots, x_{(k)}) =$$

$$\frac{n!}{(n-k)!} \prod_{1 \le i \le k} \lambda e^{-\lambda x(i)} [e^{-x(k)}]^{n-k} =$$

$$[(n\lambda)e^{-n\lambda x_{(1)}}] \cdot [(n-1)\lambda e^{-(n-1)\lambda(x_{(2)}-x_{(1)})}] \ldots$$

$$[(n-k+1)\lambda\, e^{-(n-k+1)(x_{(k)}-x_{(k-1)})}] .$$

This is the same result as is obtained by assuming that each interarrival time $\tilde{t}_{(i)} - \tilde{t}_{(i-1)}$ between failures is negative exponentially distributed with arrival rate $(n-i+1)\lambda$, as done by Musa (1975). Because of the assumption that errors are of varying degrees of difficulty, one obtains

$$f_{\tilde{t}_{(1)}, \ldots, \tilde{t}_{(k)}}(x_{(1)}, \ldots, x_{(k)}) = \frac{n!}{(n-k)!} \prod_{1 \le i \le k} \frac{\alpha^{\beta}\beta}{(x_{(i)}+\alpha)^{\beta+1}} \cdot$$

$$\frac{\alpha^{\beta(n-k)}}{(x_{(k)}+\alpha)^{\beta(n-k)}}$$

$$= \frac{\alpha^{k\beta}\,\beta^{k} n!}{(n-k)!} \; \frac{\alpha^{\beta(n-k)}}{(x_{(k)}+\alpha)^{\beta(n-k)}}$$

$$= \frac{n!}{(n-k)!} \; \frac{\alpha^{n\beta}\,\beta^{k}}{(x_{(k)}+\alpha)^{\beta(n-k)} \prod\limits_{1 \le i \le k} (x_{(i)}+\alpha)^{\beta+1}}$$

for the Gamma-Exponential model, and for the Gamma-Rayleigh model:

$$f_{t_{(1)}, \ldots, t_{(k)}}(x_{(1)}, \ldots, x_{(k)}) = \frac{n!}{(n-k)!} \prod_{1 \le i \le k} \frac{x_{(i)} \alpha^{\beta} \beta}{\left(\frac{x^2_{(i)}}{2} + \alpha\right)^{\beta+1}} \cdot \left[1 - F_{\tilde{t}}(x_{(k)})\right]^{(n-k)}$$

which cannot be simplified because $F_t(x)$ has no closed form for arbitrary x.

A diagram depicting the proposed model is given in Figure 6. Errors E_1 through E_8 are introduced unknowingly into a program at the time of its development. Associated with each error E_i is a difficulty index λ_i that serves as the parameter of the distribution for $\tilde{t}_i$, the time until E_i is discovered. Just as the introduction of errors into the program is a random process, so is the difficulty of each error so introduced. Hence the parameter $\tilde{\lambda}_i$ is also modeled as a random variable. The mixture of these distributions serves as the underlying distribution for the order statistics $\tilde{t}_{(j)}$. Each $t_{(j)}$ is the observation available for statistical analysis.

Data Domain Models

Assumptions and Terminology

One may loosely describe data domain modeling as an approach in which the form of testing is as important as, or more important than, the duration. Data domain approaches consider explicitly the input domain of the software and how it can be partitioned into relevant classes better to assess system reliability. Nelson (1973) and

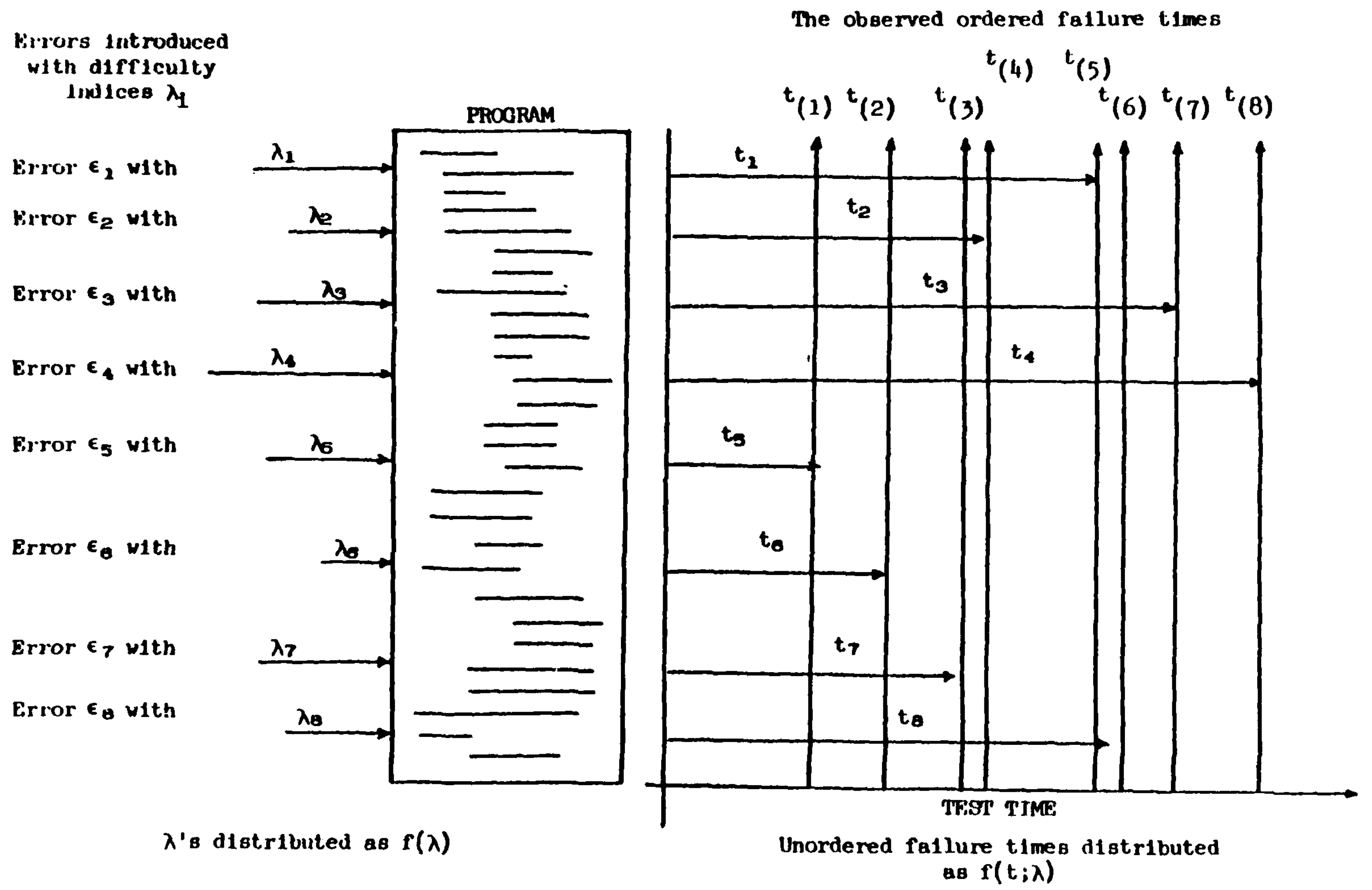


Figure 6. Error discovery in the proposed model

others have proposed partitioning the input domain D into classes D_i, for which every member of D_i executes the same program statements in the same order. Furthermore, D can be dichotomized into sets D^c, for which every element yields a correct computation, and D^e, for which every element leads to an improper program output. Let D_i^c and D_i^e be similarly defined as the correct and incorrect subsets of each path partition D_i. Nelson (1973) defined a program's reliability R as

$$R = 1 - \frac{|D^e|}{|D|} = \frac{|D^c|}{|D|} .$$

where $|A|$ denotes the cardinality of set A. In the absence of the knowledge of the precise distribution of the inputs, R can be interpreted as the probability that the program will execute correctly on any given run.

A simple reliability estimator can be derived by successive executions of a program. Lipow (Thayer et al., 1976) has shown that a sample of n inputs yielding $\hat{n}_e$ errors yields an unbiased estimator of R as

$$R = \frac{\hat{n}_e}{n}$$

when inputs are randomly chosen from D and $P_r(d \in D) \ll 1$ for all d.

The data domain approach can be better characterized by considering how each input dictates the computation of the program's function f. As outlined in previous sections, a program can be represented as a collection of segments each of which computes a partial function f_i. An input data point d forces the evaluation of the function

$$f_{i_1}(f_{i_2}(\cdots f_{i_k}(d))\cdots)$$

by dictating the sequence of segments that will be executed. Whereas a time-domain approach treats only the frequency and duration of program computation, data domain models consider the relationship of input data class to program output.

<u>The Mathematical Theory of Software Reliability</u>

Nelson extended the data domain reliability models to a sufficiently general body of formalism so that the results are termed the <u>mathematical theory of software reliability</u> (MTSR) (Thayer et al., 1976). Consider again a program that computes the function $f: D \rightarrow O$ by mapping input data to output results. The input set D of cardinality N is partitioned into a fault-producing input subset D^e and the set of data inputs, D^c, which faithfully produce the outputs desired; their cardinalities are expressed as N^e and N^c, respectively. The difference between P and a correct program P* is the manner in which f differs from the desired function f* when applied to members of D^e.

The MTSR attempts to formalize these notions in the following way. Assume that during operational use the program will be employed in a manner defined by the distribution of inputs. Assuming a finite input data space D, a mass function may be defined; each point $d_j \in D$ is assumed to be chosen with probability P_j. This set of P_j's is termed the operational profile. Letting the characteristic variable y_j indicate the membership of d_j in $D^c(y_j=0)$ or in $D^e(y_j=1)$, then

$$q = \sum_{1 \leq j \leq N} P_j y_j$$

denotes the probability that an operational run of P will result in execution failure. Conversely, system reliability may be defined as

$$R = 1 - q$$

or the probability of correct execution in the operational environment. A discrete analogue to the time-domain reliability function is

$$R_k = (1 - q)^k$$

which represents the probability of k successive correct runs.

The reliability function may be further investigated by considering test data distributions other than the operational profile $\{P_j\}$. Let

$$\rho_i = \sum_{j:d_j \in D_i} P_j = \rho_i^c + \rho_i^e$$

where $$\rho_i^c = \sum_{j:d_j \in D_i^c} P_j \quad \rho_i^e = \sum_{j:d_j \in D_i^e} P_j$$

denote the probabilities associated with drawing an arbitrary (ρ_i), a fault-producing (ρ_i^e), or a "correct" (ρ_i^c) data input point from partition D_i. A sample of n data test points are to be applied with n_i of the n corresponding to the i^{th} partition D_i. Assuming that a predetermined sample size n_i of test points from partition D_i are to be drawn, the outcome of the whole sample (i.e., test ensemble) may be denoted by $\{e_i\}$, the number of test points in the i^{th} class leading to execution failure of the n_i applied. Hence any sequence of n_i points sampled from D_i has an associated point probability of

$$\frac{(\rho_i^e)^{e_i}(\rho_i^c)^{n_i-e_i}}{\rho_i^{n_i}} .$$

Because the sampling distribution $\{\rho_i\}$ need not reflect the operational profile $\{P_j\}$, any reliability estimate derived will not necessarily be representative. A reliability estimate may be developed by calculating the ratio of program faults to successful executions. The resulting estimate will be

$$\hat{R} = 1 - \sum_{i \in T} \frac{e_i}{n_i} \cdot \rho_i$$

the probability-weighted sum of failure ratios for each partition. T is an index set of those i's for which $n_i \neq 0$. $\hat{R}$ is evidently a biased estimator of R, as

$$E(\hat{R}) = 1 - \sum_{i \in T} E(e_i) \frac{\rho_i}{n_i} = - \sum_{i \in T} \frac{[n_i \rho_i^e]}{\rho_i} \cdot \frac{\rho_i}{n_i}$$

$$= 1 - \sum_{i \in T} \rho_i^e > 1 - q = R$$

$\hat{R}$ has bias equaling the sum of the error probabilities for those partitions unsampled. Because $\text{Var}\,[e_i] = n_i \rho_i^e \rho_i^c / \rho_i^2$

$$\mathrm{Var}[\hat{R}] = \sum_{i \in T} \frac{\mathrm{Var}(e_i)}{n_i^2} \cdot \rho_i^2 = \sum_{i \in T} \frac{\rho_i^e \rho_i^c}{n_i}$$

and the mean square error is given by

$$E[(\hat{R} - R)^2] = \mathrm{Var}[\hat{R}] = \left(\sum_{i \notin T} \rho_i^e \right)^2 .$$

A minimum variance estimator R* may be developed by appropriate choice of n_i's. Thayer et al. (1976) showed that the minimum variance estimator R* results when

$$n_i^* \cong \frac{n\sqrt{\rho_i^c \cdot \rho_i^e}}{\sum_k \sqrt{\rho_k^c \ \rho_k^e}}$$

with minimum variance

$$\mathrm{Var}(R^*) = \frac{1}{n}\left(\sum_i \sqrt{\rho_i^c \ \rho_i^e} \right)^2 .$$

Thayer et al. (1976) developed a number of confidence limit formulations used to express the likelihood that the true reliability resides within a specified interval $(\hat{R}_L, \hat{R}_U)$. An approximate confidence interval

$$\hat{R} \pm \sqrt{\hat{V} \cdot t_{(n;(1-\alpha)/2)}}$$

may be constructed about reliability estimator $\hat{R}$, with variance estimated as $\hat{V}$, assuming asymptotic normality of $\hat{R}$. A positive bias in $\hat{R}$ may lead one to employ a smaller α than would otherwise have been used.

Often one is interested only in the lower confidence limit $\hat{R}_L$. This can be computed with a one-sided t formulation as above, or by exact methods formulated by Neyman (1937). Defining the software's operational reliability as $R_i = \rho_i{}^c/\rho_i$, the software's operational reliability may be expressed as weighted average

$$R = \sum_i R_i \rho_i = \sum_i \rho_i{}^c .$$

Assuming a sample of n_i data sets is drawn from each D_i, and e_i lead to execution error an "exact" lower limit $\hat{R}_L(\alpha)$ can be given:

$$\hat{R}_L(\alpha) = n\,(1-\alpha)^{1/n} \prod_{1\le i\le k} \left(\frac{\rho_i}{n_i}\right)^{n_i/n}$$

where $n = n_1 + n_2 + \cdots + n_k$ as defined above.

Estimators of Program Error Content

A variety of other approaches has been suggested for estimating the number of errors remaining in a software package.

The simplest of these approaches is the handbook approach in which error proportion statistics are maintained by program type and testing phase. Moranda (1975), for example, cited the factor two errors per hundred object instructions as a universal error proportionality. Walston and Felix (1976) reported on the development statistics aggregated over 60 programming projects of varying sizes. Letting n_e'

denote the number of errors per thousand source lines purges during validation, and n''_e denote the number of errors per thousand source lines purged during the maintenance phase, these researche s reported the 25 percent, 50 percent, and 75 percent quartiles for n'_e and n''_e as (.8, 3.1, 8.0) and (.2, 1.4, 2.9), respectively. These and other findings appear to invalidate any assumption of a universal error content proportionality.

Other estimators involving a little more program-specific information include Halstead's (1977) error equation. The error equation is derived from assumptions regarding human information processing capability and relates to other derivations that Halstead collectively terms software science. Halstead's error content estimate $\hat{B}$, is given by

$$\hat{B} = V/3000$$

where V denotes program volume given by

$$V = N \log_2 n$$

for a program of N syntactic elements, n of which are unique. A number of studies have validated $\hat{B}$ as a reasonable estimator of error content (Funami & Halstead, 1975; Cornell & Halstead; Love & Bowman, 1976).

Mills (1970) proposed another error content estimation technique that has been further analyzed by Basin (1973). Assume that of N syntactic units composing a program, n_e are erroneous for some reason. An estimate of n_e can be derived either by seeding n_s new errors and allowing a tester to discover errors n_d of the $n_e + n_s$ present, or by comparing the errors detected by two independent testers. Assuming that the error samples are the result of simple random draws, an estimator can be derived from a maximum likelihood estimator based upon the hypergeometric distribution. The estimator is biased with both the bias and estimator variance as a function of n_e. Further details are given in Basin (1973).

CHAPTER IV

EXPERIMENTAL DESIGN AND PROCEDURES

Experimental Objectives

An experiment was designed to collect the data necessary to satisfy two experimental objectives. The basic objective of the experiment was the discovery of relationships between the quality of testing and each of the sources of testing variability that affect it. This was accomplished by an experimental design in which the testing performances of subjects were observed under varying "conditions" of alternative program types, testing approaches, and subject characteristics. It was a second objective of the experiment to acquire experimental observations to infer whether sources of variability in individual characteristics, program types, and testing methods affect the forms of distributions related to reliability and its growth over time.

Sources of Experimental Variability

Because few testing experiments have been conducted for publication, there existed an opportunity to collect new kinds of information from small debugging exercises that could better explain the relationships between program complexity, the arrival behavior of program failures (manifestation of errors), and the controlled (code review, test data selection) and stochastic (error discovery, error correction) durations of debugging activities. Because of the high cost and short duration of the proposed experiment, it was necessary to consider at length the dependence between the observational variable chosen and the mechanical considerations necessary to insure that these data were collected accurately. The experimental activities of the testing subjects had to be representative of "real" software testing, and yet be sufficiently structured that subjects could participate in data recording without undue annoyance.

The experiment was designed around three "objects" varied to allow inferences to be made regarding the nature of testing. (See Figure 7.) The experimental objects include:

- The Tester-subjects: a group of 22 professional programmers chosen to represent a universe of conventional software developers of small- to medium-scale software.
- The Experimental Programs: a set of four small (60-300 lines) programs chosen to represent a universe of conventional applications over several problem domains.
- The Testing Methods: two approaches--"white-box" and "black-box"--chosen to represent one testing approach for each of two extreme points of view.

The details regarding these experimental factors are given below.

Experimental Subjects and Individual Differences

In his analysis of a large verification experiment, Hetzel (1975) found that computing experience/education and self-confidence were significantly related to better testing performance, with the "best" of 39 subjects performing two to three times as productively as the "worst." In deference to Hetzel's findings, it was assumed that

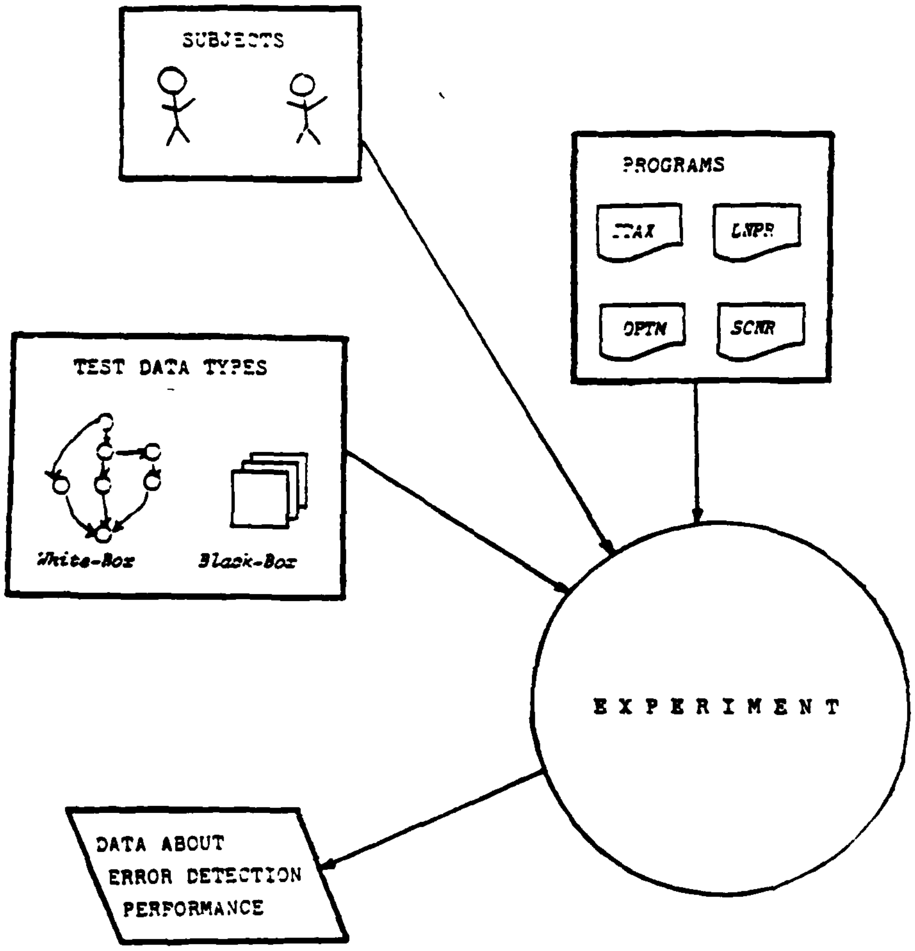


Figure 7. A pictorial overview of the experiment

the differences in individuals that affect testing performance could be estimated by obtaining values for personal attributes.

Each of 22 experimental subjects was interviewed through a questionnaire to obtain biographical data that might have a bearing on subject performance. The data requested from each subject included:

- Subject Name
- Subject Age
- Subject Sex
- General Educational Background and Degrees Earned
- Educational Background in Computer-Related Studies
- Educational Programming Experience
- Educational Background in Programming Theory and Practice
- Professional Background in the Information Sciences
- Professional Programming Experience
- Composition of Programming Experience by Programming Language
- Composition of Programming Experience by System Environment (batch/interactive)
- Subject Estimation of Proficiency in Areas Useful to Programmers
- Subject Indication of Familiarity with Theory and Practice Regarding Each Experimental Program

Subjects were asked to include in their educational background any high school or college coursework, as well as training schools, company-sponsored educational programs, and extended professional seminars. Professional experience included full- and part-time employment as well as independent development of skills. A self-assessment of proficiency was asked of each user in categories ranging from academic fields of study to computing skills; responses were chosen from a five-point ordinal scale.

In estimating his proficiency in the theory and implementation of each of the four problem classes represented by the experimental programs, a subject could choose among five ordered levels of familiarity with the problem. Copies of the questionnaires used are included in Appendix A.

Experimental Programs and Their Characteristics

It is a natural and common belief among computing practitioners and researchers that some programs are more difficult to program and debug than others. Thayer et al. (1976) and others indicated that the propensity to err in programming is related to the "size" of the task being attempted and the number of paths of execution possible within an implementation (i.e., program performing the task). One may denote "size" as computational content, the degree of arithmetic and symbolic processing represented by a program. Logical complexity is the usual nomenclature used to denote the degree to which a program departs from simple, sequential, unconditional execution.

It was hypothesized that the degree of computational content and logical complexity associated with a program would materially affect the performance of experimental subjects attempting to find program errors. Hence an integral part of the experimental design was the choice of four programs, representing each combination of high and low degrees of computational content and logical complexity. The logical complexity of a program was measured by McCabe's (1976) complexity measure and by TRW Systems' complexity metric (Thayer et al., 1976). The degree of a program's computational content was assessed by a method inspired by Halstead's (1977) metrics, which are defined in terms of the number of program operators and operands. The formulae for these metrics and calculations for each experimental program are presented in Appendix B.

So that this investigator would have a feeling for the range and sensitivity of the metrics, these measures of logical complexity and computational content were computed for a sample of programs chosen from an available program library. In this way, upper and lower bounds were defined for low and high settings, respectively, of each metric. Details are given in Appendix B.

All other decisions regarding the form and substance of the experimental programs were made in favor of simplicity and representativeness. For reasons of convenience, accessibility, and ease of experimental administration, the Keck Management Science Center of the University of Southern California School of Business was chosen as the experimental site. The Center's resident time-shared minicomputer system (Hewlett-Packard 2000 Access System) was selected as the experimental computing resource. All programs were written employing a small subset of the BASIC language that most resembles other BASIC implementations as well as semantically similar statements in COBOL, FORTRAN, PL/I, and ALGOL.

The precise choice of programs to be used in the experiment was a time-consuming, iterative procedure. To ensure that the results of this research would be generalizable to a large population of program types, a large set of diverse programming problems was considered, from which four programs were chosen for use in the experiment. Each program candidate, in turn, was formally specified and coded in BASIC. The metrics were computed for each successively coded program. If a program produced unambiguously high- or low-valued measures for each metric, it was included as a candiate for one of the four experimental programs.

The experimental programs chosen were deemed as mutually diverse as possible. They are described below and in Appendix C.

- ITAX: a 123-statement program that computes state and federal income tax in conformity to a set of prescribed rules for income and deduction admissibility. ITAX is <u>low</u> in both computational content and logical complexity.
- OPTM: a 70-statement program that solves for the local optima of a fifth-degree polynomial. OPTM is <u>high</u> in computational content and <u>low</u> in logical complexity.
- SCNR: a 260-statement program that determines and includes the symbols of a fictitious programming language. SCNR is <u>low</u> in computational content and <u>high</u> in logical complexity.
- LNPR: a 205-statement program that solves linear programming problems by either primal or dual simplex algorithms. LNPR is <u>high</u> in both computational content and logical complexity.

Test Data Types

Of the test data generation methods discussed in Chapter I, two were chosen to determine whether any differences in debugging performance could be attributed to differences in the "program stimuli" (program output on failure) generated by alternative test set methodologies.

"White-box" test data were generated by following the procedures outlined by Meyers in The Art of Software Testing (1979). A program control graph was constructed and input data points were devised to ensure that all of the major program execution paths were taken. A "black-box" test data set was also constructed in conformance with Meyers: test data points were included that verified conformance to explicit rules of program behavior indicated by the program's specification. Specific components of each test data set generation category (black-box and white-box) are listed below. Descriptions of each testing approach were given in Chapter I.

White-Box	Black-Box
Path-testing	Specification-based testing
Branch-covering	Special values testing
Structural testing	Functional testing

A description of the steps taken to develop white-box and black-box test data sets is detailed in Appendix D.

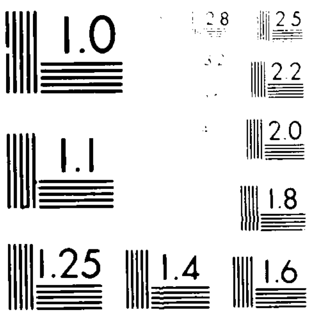
Program Errors

In attempting to discover the factors affecting program debugging, this and all prior experiments (Rubey et al., 1975; Howden, 1978; Hetzel, 1975) found it necessary to use real programs containing "naturally occurring" errors. In an ideal experiment each subject would develop a program from a common specification and be observed during his or her pursuit of errors. However, the cost involved in conducting even a moderate-sized experiment of this kind would be

AD-A110 366 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT O--ETC F/G 9/2
A STUDY OF FACTORS AFFECTING SOFTWARE TESTING PERFORMANCE AND C--ETC(U)
JUN 80 J L BAHR

UNCLASSIFIED NL

2 OF 3
AD A
110366


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

prohibitive unless the resulting programs were of some commercial use. It is precisely this high cost of experimentation that motivated Boehm (1973) to implore the computing community to record data on programming projects so that the results of comparable development activities could be analyzed in a post-facto "experiment."

To insure a comparability of stimuli to each experimental subject, it was necessary that each debugging participant face not only the same programs, but the same population of unidentified errors. After selecting the set of experimental programs, this investigator debugged each program and subjected each program to a collection of white-box, black-box, and ad hoc test data sets. Each error found was numbered and categorized as _computational_, _logic/sequencing_, or _data handling_.

These error categories were described in Chapter II; a listing of the specific errors found is given in Appendix E.

Experimental Design

Design Objectives

Each decision concerning the experiment was influenced by one of two research motivations. The primary influence on the structure of the experiment was the necessity that analysis be able to disambiguate the effects of various factors on observed performance. For this reason a fractional factorial experimental design dictated the assignment of programs and test data to subjects. Although the effects of individual differences could not be controlled, it was hoped that the differences among the performances of subjects under similar conditions could be partially explained by the (measured) differences in background and personality. Some possible influences were controlled: subjects faced identical programs on identical days of the week for the same duration using an identical system.

The assignment of subjects to levels of experimental variability is shown in Table 4. The table shows, for each subject, the group of which each subject was a member, the order in which the subject was required to debug the programs, and the type of test data sets provided for each experimental program. Debugging order is indicated by a permutation of the experimental programs' initials (I[TAX], L[NPR], O[PTM], S[CNR]), and test data type is designated as black-box (BB) or white-box (WB). The experiment produced 88 outcomes, resulting from 11 replications of the 2 x 2 x 2 fractional factorial design.

The second general research motivation was the desire to provide better answers to questions about important activities in software development and to test some of the assumptions prevalent among software researchers and practitioners. Some of these questions were:

- Do individual differences matter--age, sex, education, or experience with similar problems or similar programming environments?
- Do the test data make a difference in the rate or nature of errors found, or become important merely when verifying program correctness?
- Are some programs inherently more difficult to comprehend? Are those differences quantifiable?
- Are some types of errors more difficult to detect? Is debugging a random sampling of remaining errors?

Table 4

Assignment of Factor Settings to Subjects

			Test type by subject/program				
			ITAX	LNPR	OPTM	SCNR	Program
			Low	High	Low	High	Logical complexity
			Low	High	High	Low	Computational content
Subject number	Group number	Debugging order	One hour	Three hours	Two hours	Three hours	Time allowed
1	1	LOIS	BB	BB	WB	WB	
2	1	OLSI	BB	BB	WB	WB	
3	1	OSIL	BB	BB	WB	WB	
4	1	LIOS	BB	BB	WB	WB	
5	1	SOIL	BB	BB	WB	WB	
6	1	OLIS	BB	BB	WB	WB	
7	2	ISOL	WB	WB	BB	BB	
8	2	LSOI	WB	WB	BB	BB	
9	2	ISLO	BB	BB	WB	WB	
10	2	ILOS	BB	BB	WB	WB	
11	2	SLOI	WB	WB	BB	BB	
12	2	ILSO	WB	WB	BB	BB	
13	2	SILO	WB	WB	BB	BB	
14	2	LSIO	WB	WB	BB	BB	
15	2	SLIO	WB	WB	BB	BB	
16	3	IOLS	BB	BB	WB	WB	
17	3	SOLI	WB	WB	BB	BB	
18	3	OSLI	BB	BB	WB	WB	
19	3	OISL	WB	WB	BB	BB	
20	3	SIOL	WB	WB	BB	BB	
21	3	LISO	BB	BB	WB	WB	
22	3	OILS	WB	WB	BB	BB	

The experiment was designed so that at least some of these questions could be formulated in terms of statistical tests. It was hoped that for the remaining questions, the results of this study would at least provide the basis for an informed opinion.

Notation for Factors and Observations

The experiment was designed to determine which personal and environmental factors affect the detection and correction of program errors. The recognition of erroneous program statements was considered a significant event, and data collection forms were designed for recording the specific errors found and the times of their discovery.

A variety of influences was expected to affect error discovery times:

- The programs in which the errors were found
- The test data set types that subjects were permitted to use
- The backgrounds and personality traits of the subject.

The program characteristics that were expected to influence performance were identified as the degree of _logical complexity_ and _computational content_. Each experimental program was chosen because it represented one of four combinations of two factors at two levels. Moreover, for each program, each subject was instructed to use one of two test data sets--_white-box_ or _black-box_. Hence each observation of a detected error occurred in the context of one of eight experimental states. Each subject was given a different ordering in which to debug the experimental programs, in hopes of controlling any learning or degradation effect upon performance. Two basic types of performance measures were determined for each experimental quantum involving a subject and a program: the <u>number and kinds of errors found</u> and the time distribution of these events.

Let E_{jklm} denote the number of errors found by subject m, where

$$j = \begin{cases} 0 & \text{implies that the errors were found in a program of low logical complexity (ITAX or OPTM)} \\ 1 & \text{implies that the errors were found in a program of high logical complexity (SCNR or LNPR)} \end{cases}$$

$$k = \begin{cases} 0 & \text{implies that the errors were found in a program of low computational content (ITAX or SCNR)} \\ 1 & \text{implies that the errors were found in a program of high computational content (OPTM or LNPR)} \end{cases}$$

$$\ell = \begin{cases} 0 & \text{implies white-box test data were employed} \\ 1 & \text{implies black-box test data were employed} \end{cases}$$

Furthermore, denote error counts mnemonically, as:

$I_m^w \equiv E_{000m}$ (ITAX, white-box),

$I_m^b \equiv E_{001m}$ (ITAX, black-box),

$P_m^w \equiv E_{010m}$ (OPTM, white-box),

$P_m^b \equiv E_{011m}$ (OPTM, black-box),

$S_m^w \equiv E_{100m}$ (SCNR, white-box),

$S_m^b \equiv E_{101m}$ (SCNR, black-box),

$L_m^w \equiv E_{110m}$ (LNPR, white-box),

$L_m^b \equiv E_{111m}$ (LNPR, black-box).

For analyzing error-detection times and interarrival intervals, let t^{i}_{jklm} denote the time to the discovery of error number i for subject m, where j, k, and l indicate factor settings as described above. The superscript indexes the errors in an arbitrary order. Let $t^{(i)}_{jklm}$ denote the time to the i^{th} error detected and represent an order statistic. The number of errors known to be present in each program is fixed and denoted as:

$N_{00} = N_I \equiv$ the number of errors known to be present in ITAX,

$N_{01} = N_P \equiv$ the number of errors known to be present in OPTM,

$N_{10} = N_S \equiv$ the number of errors known to be present in SCNR,

and $N_{11} = N_L \equiv$ the number of errors known to be present in LNPR.

All random variables will be denoted by the use of tildes; hence, where $\tilde{x}$ may denote an observation prior to sampling, x represents the sample outcome. Sums of indexed variables will be denoted by the replacement of the index of summation by an asterisk. Hence,

$$x_{jk*m} = \sum_{0 \le \ell \le 1} x_{jklm}$$

and

$$x_{*k*m} = \sum_{0 \le j \le 1} \sum_{0 \le \ell \le 1} x_{jklm} .$$

Means will be denoted similarly with the addition of a bar. Hence,

$$\bar{x}_{j*lm} = \sum_{0 \le k \le 1} \frac{x_{jklm}}{2}$$

and

$$\bar{x}_{j**m} = \sum_{0 \le k \le 1} \sum_{0 \le \ell \le 1} \frac{x_{jklm}}{4} .$$

Experimental Procedures

Experimental subjects were recruited from industrial and academic environments in the Los Angeles area. To be considered as a prospective experimental subject, each candidate was interviewed to determine if he would be representative of the population of programming professionals. The criteria for subject selection were that:

- The subject had been employed in a programming capacity for at least 12 months in the previous five years.
- The subject was currently employed in a computer-related field that required at least occasional programming.
- The subject had had at least a minimal exposure to the BASIC programming language, and substantial experience with a common procedural language (e.g., FORTRAN, COBOL, or ALGOL).
- The subject had had some experience with interactive, terminal-oriented programming systems.

Twenty-two subjects were engaged for the experiment and assigned one of three dates for participation in the experiment. Each subject was given a packet containing:

- An introductory letter outlining the purpose and scope of the experiment.
- A questionnaire to determine educational, biographical, and professional backgrounds.

- A set of four specifications describing the purpose, inputs, outputs, and procedural approach for each of the experimental programs ITAX, LNPR, OPTM, and SCNR.
- Activity logs and program modification logs upon which to record progress during the debugging of each program.
- An experimental timetable and map to the experimental site.
- A copy of the rules for Keck Center and the procedures necessary for interacting with the Hewlett-Packard 2000 system.
- A synopsis of the Hewlett-Packard system commands and the BASIC language statements supported.

Copies of these documents can be found in Appendices A, C, and F.

The set of subjects was split into three approximately equal groups; each group met on a different Sunday, during which the computing center was closed to all others. Each session was identical in format, starting at 8 a.m. and finishing at 8:30 p.m. with scheduled breaks for orientation and meals.

During the orientation session the subjects received a review of useful system commands and anomalies of Hewlett-Packard BASIC that had unavoidably been introduced within the experimental programs. All subjects were instructed on the use of the data collection forms and given the time durations allowed for each experimental program. Each subject was told:

- the system account number and password specifically assigned to him or her,
- the order in which the experimental programs were to be debugged, and
- for each program, whether white-box or black-box test data were to be employed.

In each subject's program library were the programs and data files necessary to conduct the experiment. These included:

- The four completely debugged experimental programs named ITAX, LNPR, OPTM, and SCNR; subjects were able to run these programs, but were prevented by the system from listing them.
- The four undebugged experimental programs named ITAXØØ, LNPRØØ, OPTMØØ, and SCNRØØ; these programs were constructed by reinserting the original errors within ITAX, LNPR, OPTM, and SCNR; subjects were permitted to modify, run, or list these versions.
- Eight sets of test data sets--a "white-box" and "black-box" set--for each of the four programs.

Subjects were asked to keep track of the time spent on each program and required to record on activity logs the start and stop times of each debugging activity. Activities included:

- Code Review/Error Detection: any activity in which errors are being sought by reading the program listing and comparing it to program specifications.
- Error Correction: any significant time duration needed to formulate the "fix" required to eliminate an identifiable program error.
- Terminal Work: any clerical activity involving use of the computing system--making program changes, retrieving and saving program versions, obtaining fresh program listings.
- Test Data Set Development: developing test data to exercise the experimental programs; subjects were not permitted to develop their own test data unless they had debugged a program well enough that it ran, without error, all prescribed white-box or black-box test sets.

- Break: any nontrivial mental or physical departure from one of the above activities.

Upon detecting an error, a subject was requested to post the statement number(s) affecting the error, the time of discovery, the corrective action, and any comments that may have been of interest to this investigator. This investigator and a proctor made periodic checks on the progress of each subject and reviewed the data collection sheets to insure that activity was being properly documented. This investigator was available during each session to explain perceived ambiguities in the specifications, aid those who had forgotten system commands and procedures, and otherwise minimize the time that subjects needed to spend in nondebugging activities.

CHAPTER V

DATA ANALYSIS

Basic Results and Descriptive Statistics

The 22 subjects constituted a diverse set of software professionals. Ten subjects worked full time for a company whose product was not computer-related, and 7 subjects were regularly employed to produce software systems for resale. The remaining 5 subjects were academicians currently teaching computer studies, and who had had significant programming experience. Ages of the subjects ranged from 19 to 53. Median age was 28 and mean age was approximately 31. There were 5 female subjects of the 22. A histogram of the subjects' ages is shown below in Figure 8.

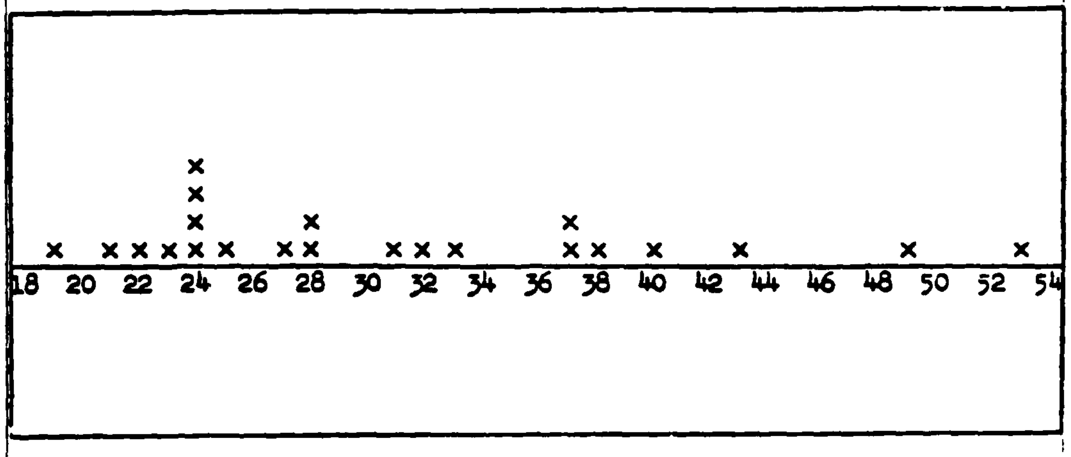


Figure 8. Distribution of subjects' ages

Educational attainment was measured by five increasingly specific variables:

- Highest degree earned, an ordinally scaled variable with permissible values (0) none, (1) high school, (2) bachelor's, (3) master's, (4) all-but-dissertation, or (5) doctorate.
- Full-time years of schooling, which could include training courses.
- Courses in computer-related subjects, measured in "standard units"; a quarter-unit was assigned 2 standard units and a semester-unit was assigned 3 standard units.
- Courses requiring a significant degree of programming, measured in standard units.
- Courses in programming theory and practice, measured in standard units.

The subject set represented the full spectrum of educational background. Subjects' years of schooling ranged from 9 to 26 years, and all degree levels were represented. The diversity of training in computer studies was similarly wide; three of the subjects had virtually no formal training in the field, while three had over 30-semester units of computer studies. Educational background in programming theory and practice was particularly dichotomous, with 7 subjects having taken no courses, whereas 11 subjects had taken at least 4 courses. Histograms of subjects' educational background are shown in Figure 9.

The first histogram indicates the number of subjects within each of the degree attainment categories (next to each "X" signifying a subject, is the number of years of schooling obtained by that subject). The three remaining frequency distributions show where each

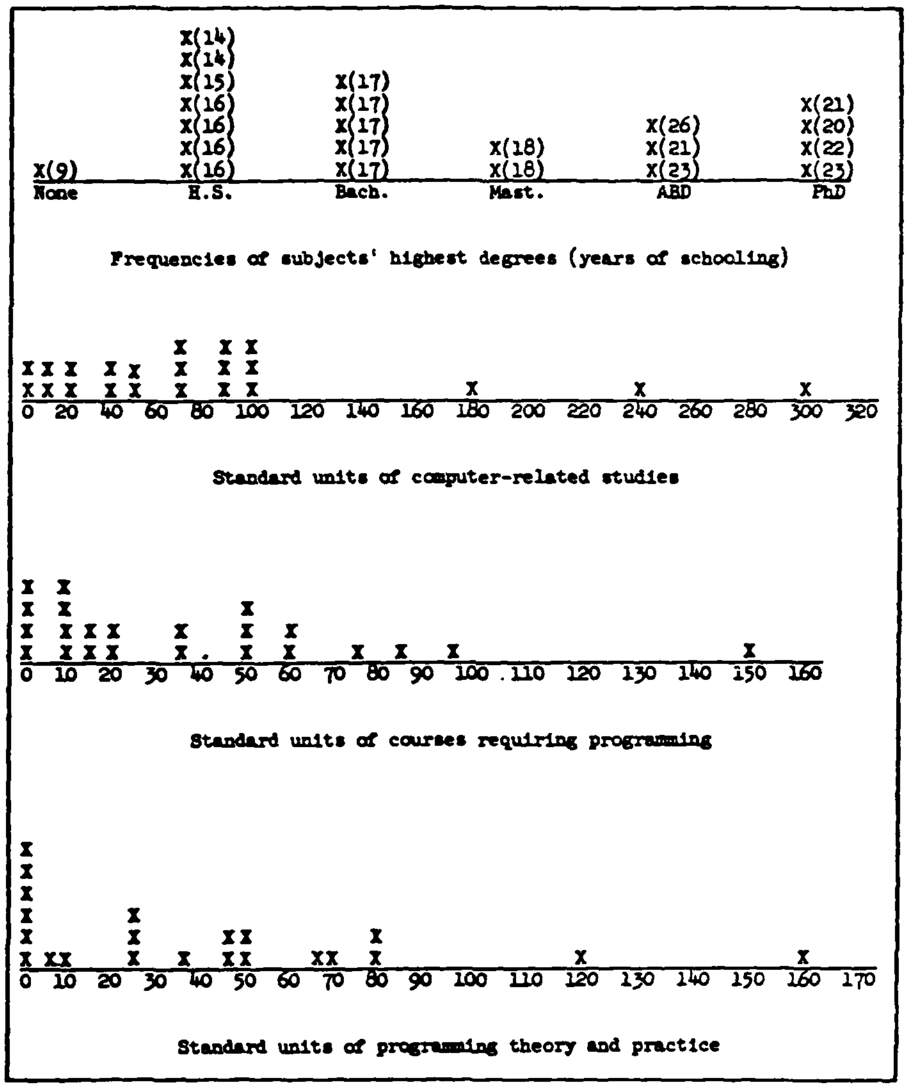


Figure 9. Sample distributions of subjects' educational background

of the 22 subjects reside on educational scales measured by the number of standard units (½ semester unit) taken in different subject categories.

In Figure 10, the sample distribution of subjects' professional experience is displayed. Any endeavor in the computing field was included as general experience-programming, systems analysis, project management, or teaching. Consistent with the wide disparity among subject ages, the range of general professional experience was similarly broad--from less than 10 man-months to over 160. The median was approximately 55 man-months, indicating a mild skewing to the right. Professional experience in programming was significantly more skewed, with the bulk of the subjects having worked less than 40 man-months in programming activities. The left-shifted median of approximately 20 man-months reflects the tendency of subjects to shift to analysis and management roles with experience, and as the non-programming experience of the academicians in the sample.

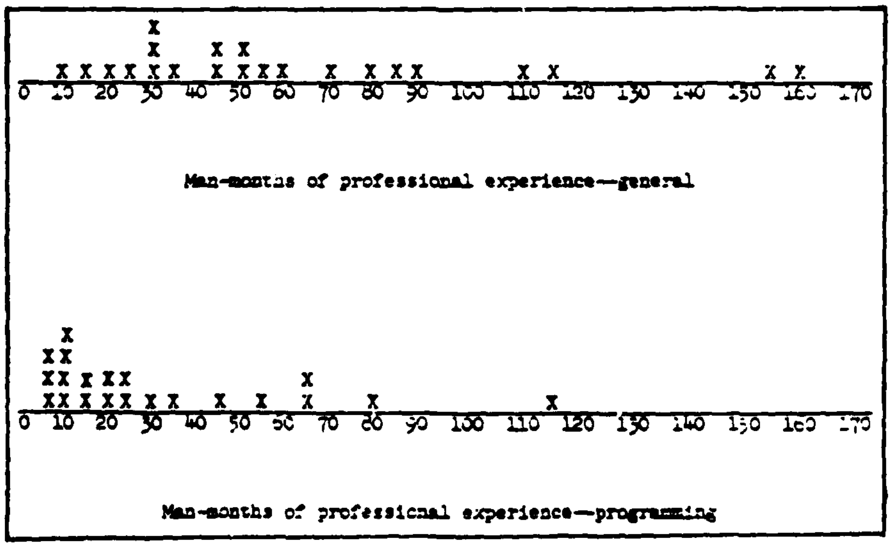


Figure 10. Sample distributions of subjects' professional experience

An attempt was made when selecting subjects to insure that each subject had programmed in an environment similar to that of the experiment. Overall subject familiarity with interactive systems was high, and most subjects indicated that their current programming was performed on such systems. The vast majority of subjects, however, had programmed very little in BASIC, although all subjects had substantial experience in similar procedural languages.

Unfamiliarity with BASIC had no apparent effect on the subjects. There were virtually no questions regarding the language during the experiment, and no evidence of its misuse upon analysis of the subjects' work. Sample distributions concerning environmental factors are given in Figure 11.

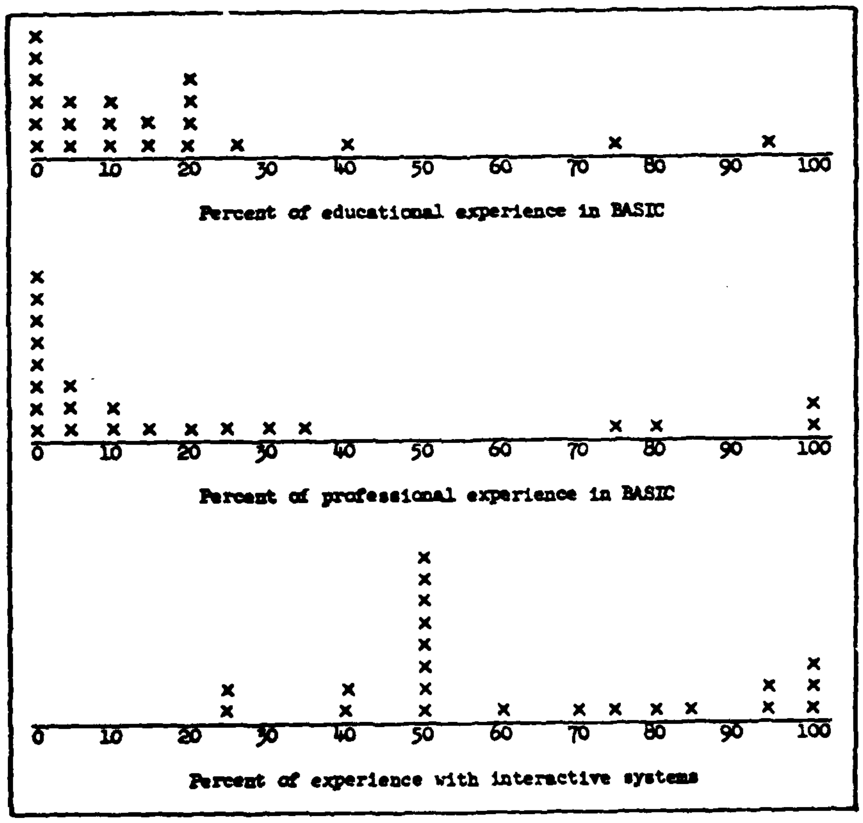


Figure 11. Sample distributions of environmental variables

The final question on the experimental questionnaire asked the subjects to assess their knowledge and proficiency in 12 areas. This question was motivated by Hetzel's findings in a related study (1975) that confidence (as measured through self-assessment) correlated well with testing performance. Hetzel asked a relatively homogeneous group to rate themselves with respect to the other experimental subjects. In this study, the subjects were (non-specifically) asked to rate themselves. It was assumed that this nonspecificity would induce the subjects to rate themselves against their perception of the general knowledge and proficiency levels of the population of computing professionals.

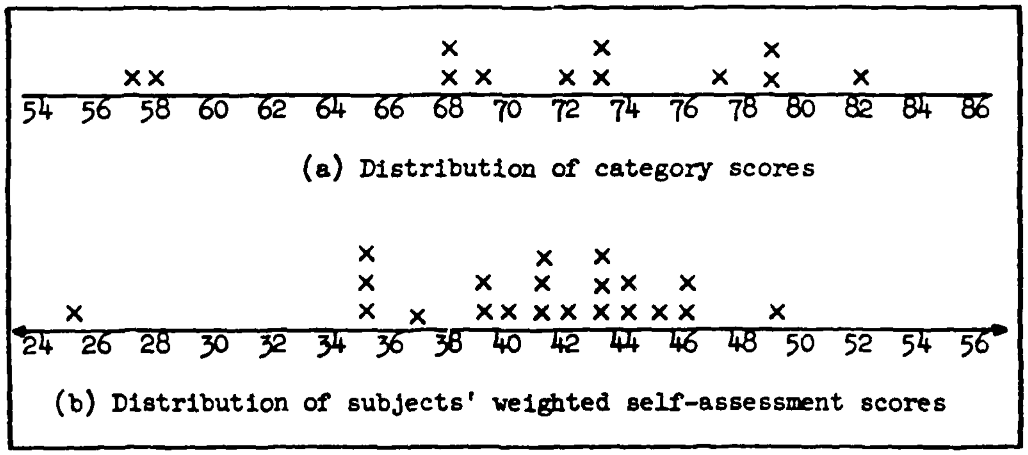
The group of subjects responded, as a whole, with a reasonably high self-assessment. The most likely response for every category but one (operations research) was AVERAGE or STRONG, and the response medians were closer to STRONG. Questions regarding subject proficiency in programming tasks and knowledge of data processing principles were responded to with the highest self-assessment. Other programming-related skills were generally responded to with positive self-assessment as was proficiency in mathematics. The two application areas (probability/statistics and operations research) resulted in the lowest self-assessment responses.

Table 5 shows a breakdown of responses by proficiency category. A measure was constructed by a linear weighting of each response on a scale of 1 to 5, resulting in a weighted self-assessment score for each subject; a similar measure of overall sample self-assessment was also calculated for each category. The resulting distribution of category

Table 5

Breakdown of Self-Assessment Ratings by Category

	Ratings					Category Score
	1	2	3	4	5	
Data processing	0	1	7	11	2	77
Computer science	2	1	8	9	1	69
Systems analysis	0	2	8	10	1	73
Systems design	1	4	5	11	0	68
Program design	0	0	5	13	3	82
Program writing	0	1	7	9	4	79
Program debugging	0	1	8	7	5	79
Operations research	4	6	5	4	2	57
Probability/statistics	4	4	7	5	1	58
Mathematics	0	3	9	6	3	72
File handling	0	2	9	8	2	73
Algorithms	1	4	8	5	3	68
Rating Frequency	12	29	86	98	27	



(a) Distribution of category scores

(b) Distribution of subjects' weighted self-assessment scores

Figure 12. An analysis of self-assessment

scores is shown in Figure 12(a). The sample distribution of subject self-assessment scores is shown in Figure 12(b). The subjects showed little variability in their overall self-assessment, as is demonstrated by the clustering of most subject scores in the interval from 35 to 45 on a scale from 12 (all _very weak_ responses) to 60 (all _very strong_ responses). The vast differences in the experience levels and educational backgrounds of the subjects makes this relative uniformity in self-assessment somewhat surprising. It is conjectured that optimism and confidence may have as much influence on a subject's self-assessment as does the actual proficiency level developed by the subject. In the academic areas (mathematics, computer science, probability/statistics, operations research, and algorithms), it is conjectured that education makes one aware of one's ignorance of the area at a rate approximating one's mastery of the topic. In support of this conjecture, it was found that some subjects with little formal training ranked themselves equal to subjects with graduate work in an area.

After the subjects had studied the program specifications each was asked to specify his or her degree of familiarity with the principles underlying the application dealt with by each program (theory), and to indicate the extent of experience with programming similar applications (practice). A summary of the responses is given in Table 6. Subjects generally expressed a greater familiarity with the theory than with the techniques of implementing these programs. As to be expected, subjects were most familiar with the theory and practice of programs similar to ITAX, an income tax calculator. OPTM, the most specialized of the problems received the most negative responses, indicating lack of exposure to the problem. SCNR and LNPR dichotomized the subject sample somewhat. Those with computer science backgrounds or work experience in certain system programming areas were certain to have seen applications like SCNR, whereas all others most likely would not have. The theory behind LNPR (a linear programming code) had most likely been exposed to those with a business school educational background. This proved to be so, judging from many subjects' expressed familiarity with the theory. Most of these same individuals, however, would not generally be engaged in scientific programming work, hence the very low levels of experience in programming applications similar to LNPR.

Table 6

Breakdown of Familiarity Ratings

	Not Familiar	Not Very Familiar		Familiar	Very Familiar
ITAX-Theory	1	3	2	12	4
ITAX-Practice	4	4	4	7	3
LNPR-Theory	3	4	7	6	2
LNPR-Practice	9	7	4	0	7
OPTM-Theory	4	8	4	6	0
OPTM-Practice	11	7	0	3	1
SCNR-Theory	4	3	4	7	4
SCNR-Practice	8	5	2	4	3
Totals	44	41	27	45	18

A synopsis of the questionnaire responses is given in Table 7. Information regarding type, measurement scale, unit of measure, and descriptive statistics is given for each variable represented by or calculated from responses. Since each variable is at least ordinally scaled, the response or response category associated with the lowest and highest values is given under SAMPLE RANGE. Measures of central tendencies have been computed where appropriate. Means have been computed for some ordinally scaled variables under the assumption of equal intervals. Two variability measures have been calculated: sample average deviation and sample standard deviation. Where these measures have been calculated for ordinally scaled data, the assumption of equal intervals has been applied.

The Composition of Program Errors

In the normal course of developing each of the experimental programs, errors were inadvertently introduced into the code. These errors were found, recorded and categorized during the debugging of the programs by this investigator. Three major classes of errors were identified:

- COMPUTATIONAL. An error in which a value is improperly calculated due to an incorrect sequence of operations; these errors are the result of a missing computation, an improper expression, or a failure to recognize a machine limitation.
- LOGICAL. An error in identifying the distinct subfunctions of a program or expressing the control mechanisms used to choose among subfunctions; these errors are the result of

Table 7

Synposis of Questionnaire Variables

Category	Variable	Measurement Scale	Unit of Measure	Sample Range		Central Tendency			Variability Average	Standard
				Low	High	Mode	Median	Mean	Avg. Dev.	Std. Dev.
Age	Age	Ratio	Years	19.0	53	24	28.0	31.0	7.55	9.3
Education	Yrs. school	Ratio	Years	9.0	26	17	17.0	17.9	2.84	3.7
	Courses: C-R	Ratio	Std. units	3	299	0	70.0	79.2	51.40	74.9
	Courses: RP	Ratio	Std. units	0	[illegible]	0	21.6	38.9	28.80	36.8
	Courses: TP	Ratio	Std. units	0	[illegible]	0	24.0	40.6	33.20	43.1
	High degree	Ordinal	Degrees	none	[illegible]	H.S.	Bach.	—	--	—
Experience	General	Ratio	Man-months	8.0	165	30	48.0	62.1	34.30	43.5
	Programming	Ratio	Man-months	5.2	117	12	20.5	34.3	24.90	30.8
Environment	$ BASIC: Ed.	Ratio	Percentage	0	95	0	10.0	17.6	15.60	24.3
	$ BASIC: Prof.	Ratio	Percentage	0	100	0	5.0	23.5	25.50	33.5
	$ Interactive	Ratio	Percentage	25.0	100	50	50.0	63.0	20.80	24.0
Self-Assessment	DP principles	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 4	Rating 4	3.67	0.603	0.713
	CS principles	Ordinal	Weak-strong	Rating 1	Rating 5	Rating 4	Rating 3	3.29	0.776	1.0
	Sys. analysis	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 4	Rating 4	3.52	0.639	0.731
	Sys. design	Ordinal	Weak-strong	Rating 1	Rating 4	Rating 4	Rating 4	3.24	0.798	0.921
	Pgm. design	Ordinal	Weak-strong	Rating 3	Rating 5	Rating 4	Rating 4	3.90	0.431	0.610
	Pgm. writing	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 4	Rating 4	3.76	0.676	0.811
	Pgm. debugging	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 3	Rating 4	3.76	0.748	0.867
	Ops. research	Ordinal	Weak-strong	Rating 1	Rating 5	Rating 2	Rating 3	2.71	1.060	1.240
	Prob./stat.	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 3	Rating 3	3.48	0.961	1.150
	Mathematics	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 3	Rating 3	3.48	0.785	0.906
	File handling	Ordinal	Weak-strong	Rating 2	Rating 5	Rating 3	Rating 3	3.48	0.689	0.794
	Algorithms	Ordinal	Weak-strong	Rating 1	Rating 5	Rating 3	Rating 3	3.29	0.898	1.080
	Cat. scores	Assumed intvl.	Unitless	57.0	82	Various	73.0	71.0		7.53
	Subject score	Assumed intvl.	Unitless	25.0	49	Various	41.0	40.6		5.43
Program Familiarity	ITAX-theory	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 5	Rating 4	Rating 4	3.68	0.826	1.06
	ITAX-practice	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 5	Rating 4	Rating 3	3.05	1.140	1.33
	LNPR-theory	Ordinal	Unfmlr.-fmlr	Rating 1	Rating 5	Rating 3	Rating 3	3.00	.909	1.17
	LNPR-practice	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 5	Rating 1	Rating 2	2.00	.818	1.09
	OPTM-theory	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 4	Rating 2	Rating 2	2.55	.959	1.08
	OPTM-practice	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 4	Rating 1	Rating 1	1.91	.909	1.20
	SCNR-theory	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 5	Rating 4	Rating 3	3.18	1.180	1.37
	SCNR-practice	Ordinal	Unfmlr.-fmlr.	Rating 1	Rating 5	Rating 5	Rating 2	2.60	1.280	1.43

improper branching or statement sequencing, an *improper Boolean expression*, altogether *missing logic*, or *redundant code*.

- DATA HANDLING. An error in the selection or initialization of locations; these errors are the result of *improper variable* initialization, the use of the wrong variable name, or an error in the expression of a *subscript or substring* designation.

The enumeration of each error in all four experimental programs, as well as a more detailed explanation of error types and subclasses is given in Appendix E.

Each program error was assigned an error type (COMPUTATIONAL, LOGICAL, DATA HANDLING), and a subclass. Each error and its type/subclass designation is listed in Tables 8 through 11. In each table is shown the frequency of discovery by subjects for each known error. It is apparent that these errors were nontrivial since not one of the 62 errors was found by all 22 experimental subjects. There appears to be a dichotomy in the difficulties associated with finding errors in ITAX, LNPR, and OPTM. For these three programs, errors were found by most of the subjects, or few of them. The discovery frequencies for errors within SCNR exhibit better continuity, and appear somewhat like a discrete version of the negative exponential distribution.

Each "X" in Figure 13 represents a program error, and is shown over the scale value corresponding to the number of subjects who found it. This figure indicates the approximate distribution of error difficulties, as measured by the percentage of the subject sample who

Table 8

Errors Found: ITAX

Error Number	Number Found	Error Type	Error Subclass
1.1	6	Computational	Missing computation
1.2	7	Computational	Missing computation
2	5	Data handling	Var. by wrong name
3	16	Logical	Wrong Boolean exp.
4	15	Computational	Improper expression
5	14	Computational	Improper expression
6	12	Computational	Improper expression
7	5	Data handling	Improper init.
8	12	Logic	Wrong Boolean exp.
9	5	Data handling	Var. by wrong name

Table 9

Errors Found: LNPR

Error Number	Number Found	Error Type	Error Subclass
1	12	Computational	Improper expression
2	11	Logical	Wrong Boolean exp.
3	1	Data handling	Subscript/substring
4	5	Logical	Wrong branch/seq.
5	8	Data handling	Subscript/substring
6	16	Logical	Wrong Boolean exp.
7	1	Data handling	Improper init.
8	4	Data handling	Improper init.
9.1	2	Computational	Missing computation
9.2	2	Computational	Missing computation
10	2	Computational	Missing computation
11	4	Computational	Improper expression
12	2	Logical	Wrong branch/seq.
13	5	Logical	Wrong Boolean exp.
14.1	2	Logical	Wrong branch/seq.
14.2	2	Logical	Wrong branch/seq.

Table 10

Errors Found: OPTM

Error Number	Number Found	Error Type	Error Subclass
1	18	Computational	Improper exp.
2	21	Computational	Improper exp.
3	21	Computational	Improper exp.
4	2	Computational	Improper exp.
6	13	Logical	Wrong branch/seq.
7	6	Computational	Machine lim.
8	13	Logical	Wrong branch/seq.

Table 11

Errors Found: SCNR

Error Number	Number Found	Error Type	Error Subclass
1	0	Logical	Missing logic
2.1	1	Logical	Missing logic
2.2	1	Logical	Missing logic
2.3	0	Logical	Missing logic
2.4	0	Logical	Missing logic
3	1	Data handling	Improper init.
4	3	Data handling	Improper init.
5	3	Computational	Improper exp.
6.1	9	Logical	Missing logic
6.2	2	Logical	Missing logic
7	7	Logical	Wrong branch/seq.
8	2	Data handling	Subscript/substring
9	3	Data handling	Var. by wrong name
10	5	Computational	Missing computation
11	19	Data handling	Improper init.
12	6	Data handling	Improper init.
13	8	Data handling	Improper init.
14	0	Computational	Improper exp.
15	1	Computational	Missing computation
16	2	Logical	Missing logic
17	0	Computational	Missing computation
18	3	Data handling	Improper init.
19	4	Data handling	Improper init.
20	2	Data handling	Improper init.
21	5	Data handling	Improper init.
22	4	Data handling	Improper init.
23	6	Logical	Redundant
24	3	Logical	Redundant
25	2	Logical	Redundant

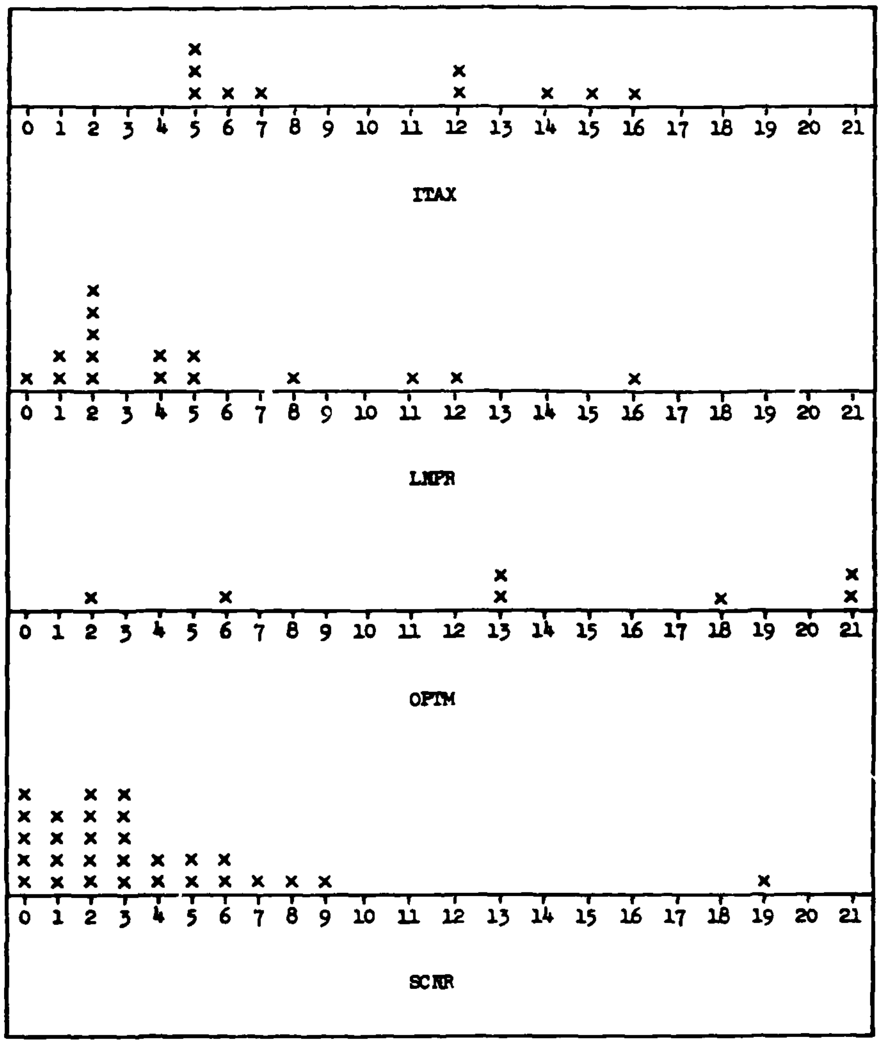


Figure 13. Frequency distribution of errors by program

discovered an error. If all errors were equally difficult, one would expect to see a symmetric histogram. The variability of subject skills may account for the spread of discovery frequencies for SCNR; the range of discovery counts is much too large for the remaining programs, however, to support the contention of similar error difficulties.

It is reasonable to expect error class frequencies to be related to the kinds of programs in which the errors are found. It would also seem possible that the discovery of particular classes of errors might be related to program characteristics. For the purpose of analyzing these possible relationships, all of the information regarding error discoveries has been summarized in Table 12. The interior table entries give the number of times that each error was found across all 22 subjects (numerator), and the number of errors represented by that entry (denominator). Totals are provided across error types, error subclasses, and programs, as well as the ratios resulting from the division of frequency of discovery by frequency of occurrence.

In reviewing the ratios for ITAX (10.5, 14.0, 5.0), LNPR (2.0, 6.14, 3.5), OPTM (13.6, 13.0, -), and SCNR (3.17, 2.75, 5.0), there is no reason to suspect a relationship between the composition of error discovery ratios and program types. Program types _do_ appear to strongly affect overall discovery ratios, considering the low discovery ratios for the logically complex LNPR (4.81) and SCNR (3.60). Across all programs, discovery ratios are highest for computational and logical errors (7.25 and 6.5, respectively), and lower for data handling errors. Correspondingly, the error population of OPTM, the program with the highest

Table 12

Breakdown of Error Types Found by Program

Error Type	Error Subclass	ITAX	LNPR	OPTM	SCNR	Totals	Ratio
Computational	Missing computation	13/2	4/3	--	6/3	23/8	2.88
	Improper expression	29/2	16/2	62/4	3/2	110/10	11.00
	Machine limitation	--	--	6/1	6/1	12/2	6.00
	Total	42/4	10/5	68/5	19/6	145/20	7.25
Logical	Wrong branch/seq.	28/2	11/4	26/2	7/1	72/9	8.00
	Wrong Boolean exp.	--	32/3	--	--	32/3	10.67
	Missing logic	--	--	--	15/8	15/8	1.88
	Redundancy	--	--	--	11/3	11/3	3.67
	Total	28/2	43/7	26/2	33/12	130/20	6.50
Data Handling	Improper init.	5/1	5/2	--	55/10	65/13	5.00
	Var. by wrong name	10/2	--	--	3/1	13/3	4.33
	Subscript/substring	--	9/2	--	2/1	11/3	3.67
	Total	15/3	14/4	--	60/12	89/19	4.68
Total		85/9	77/16	94/7	108/30	364/62	5.87
Ratio		9.44	4.81	13.43	3.60	5.87	

Note. Entries are of the form $\boxed{\text{Number found by all 22 subjects} \,/\, \text{Number in program}}$

discovery ratio, is composed mainly of computational errors, while the error population of SCNR, the program with the lowest discovery ratio, contains a higher percentage of data handling errors than any other program. The error subclasses with the highest discovery ratios, Improper Expression (11.0) and Wrong Boolean Expression (10.67) correspond to the kinds of errors which would most <u>visibly</u> disagree with the program specifications. The lowest discovery ratio, Missing Logic, is associated with the least visible of all errors since it applies to situations in which a conditional transfer is missing. Perhaps, second in their lack of visibility are errors represented by the Missing Computation category; this subclass also has the second smallest discovery ratio.

<u>Chi-Square Analysis of Error Frequencies</u>

One question of interest in this research is whether errors can reasonably be modeled as equal in difficulty. Errors of equal difficulty should have an equal opportunity to be "sampled" (by the debugging programmer) by being discovered. The distribution of successive error interarrival times could then reasonably be modeled as negative exponential, as is assumed by various Software Reliability Theorists (see Chapter IV).

A hypothesis of equal difficulties corresponds to a hypothesis of equal error discovery frequencies. This latter hypothesis can be tested by a X^2 test. Each error is assumed to have an equal chance of being included in the n_i errors detected by the i^{th} programmer. Hence, it is expected that each of the K errors will be discovered

$$e = \sum_{1 \le i \le 22} n_i/K = N/K$$

times, where N is the total number of errors found by all subjects. Let f_j denote the frequency of discovery for the j th error. Then the X^2 statistic

$$\hat{X}^2 = \sum_{1 \le j \le k} \frac{(f_j - e)^2}{e}$$

is approximately X^2 distributed with k-1 degrees of freedom. Since each error discovery is not a random sample, the chi-square test conditions are not fully met. Since, however each f_j is bounded by 0 and 22 (the number of programmers), not 0 and NK, this test is conservative since extreme values of f_j are less likely, as are large $\hat{X}^2$ values.

The results of this analysis are shown in Table 13. The hypothesis of equally likely occurrences is rejected for each of the four programs at a significant level of = .05. Furthermore, the X^2 values for LNPR, OPTM, and SCNR are decidely improbable with probabilities of occurrence of less than .01 for the stated hypothesis. A second test was performed for SCNR after removing the frequency accounting for 19 discoveries (a possible outlier); as can be seen in the table, the remaining frequencies still differ significantly from the hypothesized frequency distribution.

Tables 8, 9, 10, 11, and 12 and Figure 13 emphasized the discovery frequency of errors in an attempt to ascertain which errors appear more difficult to detect. The order in which errors are found is nearly as important in determining error difficulty, if one defines total time used that was counted toward error discovery was the minutes of code review and error correction time spent prior to error discoveries. Any time spent by the subject using the computer terminal, constructing test data or "on break" was not accumulated in these totals.

One distributional analysis of error discovery times was conducted on the reduced set of data systematically extracted from the discovery times recorded by each of the 22 sets. For each experimental program, subject error discovery times were studied and any error with less than two discoveries among all 22 subjects was dropped. Similarly, any subject who had found less than two errors was dropped from the group under analysis. The resulting matrix of discovery times generally included 60-80% of the known errors and from 10 to 16 of the original subjects.

Tables 14, 15, 16, and 17 show the reduced data sets used for the analysis of error discovery times. The columns indicate errors that had a sufficient number of errors to warrant inclusion. Each row of the table corresponds to a subject, and excludes those subjects who found less than two of the errors analyzed. The few errors found by the excluded subjects tended to be those found most often by the remaining subjects. Hence, the remaining set of subjects is assumed to be representative, if not of subject proficiency, at least of error difficulty composition. The excluded errors had been discovered at most once; dropping these errors from the estimation procedure was expected to introduce very little error in the results.

There were a number of reasons motivating this decision to reduce the analysis set. The philosophical justification for excluding "easy" errors as those generally found early and "difficult" errors as those generally found later. It is more difficult to work with error discovery orders since many individuals failed to detect some errors altogether. One means of overcoming this difficulty, for each individual, is assigning ranks to errors in the order that that individual discovered them. The errors that went undetected are then all assigned the average of the remaining ranks.

Table 13
Chi-Square Analysis of Error Discovery Frequencies

Programmer	Number of Errors	X^2 Statistic	Critical α=.05	Sig.	Value α=.01	Sig.
ITAX	10	18.98	16.92	*	21.67	
LNPR	16	66.17	26.30	*	32.00	*
OPTM	7	23.96	12.59	*	16.81	*
SCNR	29	119.03	41.34	*	48.28	*
SCNR	28	57.67	40.11	*	46.96	*

Development of Discovery Times

During the course of the experiment, the data regarding the activities of each subject were recorded. Every minute of the subjects' debugging session was accounted for and significant events were "time-stamped." These events included the start and stop times of each testing activity, as well as the times for error discovery and correction. Hence it was possible to compute the ordered error discovery times for each subject on a time scale which excluded irrelevant activities, and which was comparable for all subjects. For designated errors, the

Table 14

Error Discovery Times in Minutes (ITAX)

Error Number							
1.1/1.2	2	3	4	5/6	7	8	9
--	--	--	23	35	--	--	--
16	--	45	26	30	50	--	--
15	15	55	25	35	--	55	--
--	--	14	--	20	28	--	--
--	--	28	30	52	46	--	--
37	15	35	--	--	--	45	--
--	--	25	50	55	--	45	--
--	--	10	28	55	--	15	--
--	11	18	38	--	49	21	42
--	--	40	--	25	--	50	--
--	17	--	33	50	--	20	30
30	--	--	--	--	--	--	45

Table 15

Error Discovery Times in Minutes (LNPR)

Error Number								
1	2	4	5/6	8	9.1	9.2	11	13
97	77	102	--	118	176	--	--	107
--	25	57	45	--	--	--	--	63
96	168	193	42	--	--	--	148	--
150	--	--	130	--	--	--	--	--
140	--	--	120	--	--	--	--	--
--	110	--	65	--	--	--	--	--
130	70	--	40	--	--	--	--	--
40	184	200	78	--	--	120	159	200
75	50	85	25	95	110	120	165	70
120	83	--	15	--	--	--	--	--
103	17	--	58	--	--	--	136	--

Table 16

Error Discovery Times in Minutes (OPTM)

Error Number					
1	2	3	6	7	8
10	16	15	36	—	45
28	38	33	—	—	—
7	1	9	16	—	39
20	24	22	25	60	30
5	13	6	47	—	21
14	17	23	—	—	—
—	31	21	86	126	—
—	20	10	50	—	—
55	60	60	—	—	—
26	38	34	75	100	—
30	25	36	—	—	34
37	42	38	81	—	82
25	30	30	—	110	85
—	25	—	—	—	75
20	43	20	38	25	—
12	90	19	—	—	30

Table 17

Error Discovery Times in Minutes (SCNR)

Error Number																
4	5	6.1	6.2	7	9	10	11	12	13	18	19	21	22	23	24	27
—	—	117	—	130	—	—	20	175	148	—	—	—	—	—	—	87
—	—	85	—	145	190	—	23	—	—	—	—	—	—	—	—	—
—	—	60	—	—	—	—	—	—	—	—	150	140	—	185	45	45
—	—	—	—	—	—	133	37	52	177	—	—	—	—	149	—	—
180	180	105	105	110	165	—	70	145	—	—	—	—	—	—	—	—
124	174	20	—	25	135	—	13	—	102	33	144	—	62	114	—	—
—	—	—	—	—	—	60	—	35	40	165	20	25	85	—	—	—
—	—	—	—	—	—	—	125	—	160	—	—	35	100	—	—	—
—	—	140	—	200	—	50	100	100	100	60	70	80	—	—	—	—
—	—	90	—	105	—	—	75	35	55	—	—	5	—	—	110	—

low-performance subjects was that these subjects were not representative of the population of skilled professionals to which these results might be generalized. The practical reasons for subject exclusion was that many of the estimations and search routines exhibited undue sensitivity to a subject's lack of data. The exclusion of errors with only one observation was somewhat more justifiable, as this exclusion did not affect the number of known errors. The cost in terms of estimation accuracy of excluding a single observation was assumed to be small. Moreover, the inclusion of single points proved to make some of the estimation algorithms unusable for all other errors.

Rank Analysis of Ordered Discovery Times

The first analysis performed on the set of discovery times included a rank analysis to determine if there existed any uniformity in the order in which subjects discovered errors. The discovery times for each subject were ordered and assigned increasing ranks. Ties were handled by the mid-rank method (i.e., if the m^{th} through $(m + j)$ discovery times were identical, all were assigned the rank $(2m + j)/2$). For any subject, if exactly k errors were found, each of the n-k undiscovered errors were assigned the mid-rank $(k + 1 + n)/2$. Ranks were assigned for all 22 subjects and included all known errors.

An inspection of the ranks from subject to subject informally confirmed the hypothesis that errors were of unequal difficulty. While some errors were nearly always found early by all subjects, other errors were found by no subjects. If all errors were equally likely to be found in a given time, each permutation of discovery orders would be equally likely. Such a hypothesis of equally likely permutations can be tested by computing Kendall's coefficient of concordance. This statistic ($\hat{\tau}$) ranges from 0 to 1 and is a ratio involving the sums of ranks across all subjects among whom discordance is hypothesized. The formula for this statistic is given by

$$W = \sum_{1 \leq j \leq n} \frac{S_j^2 - 3m^2n(n+1)^2}{mn^2(n^2-1)T}$$

m = the number of rankers

n = the number of ranked objects

where $$S_j = \sum_{1 \leq i \leq m} R_{ij} \equiv \text{sum of ranks for error } j,$$

and $$\left(T = m \sum_{\ell} t_\ell^3 - \sum_{\ell} t_\ell \right)$$

where t_ℓ is the number of ties and ℓ is summed over all sets of ties. The hypothesis that is tested is one of no association among the m rankings, and the value m(n-1)W is approximately chi-square distributed with n-1 degrees of freedom (Gibbons, 1976).

Twelve hypothesis tests were conducted. For each of the four programs a group W-test was conducted. Then each of the group was spli into two groups, on the basis of whether white-box or black-box test data were employed.

One would expect the differences in test data (white-box or black-box) to modify the order in which errors were found. If the subjects' error pursuit is strongly conditioned by stimuli generated by the program, and all errors are equally likely to be provoked by a given set of test data, then different test data sets should influence error discovery order differently. To investigate the effects of test data type on error discovery order a Kendall's rank correlation measure was computed using the pairs of rank sums for each test data group. The formula for this measure is:

$$T = \hat{\tau} = \frac{S}{\sqrt{\left[\binom{n}{2} - u'\right]\left[\binom{n}{2} - v'\right]}}$$

$$u' = \sum_{\ell} \binom{u_\ell}{2} \qquad v' = \sum_{\ell} \binom{v_\ell}{2}$$

where the summation is over all sets of tied ranks for each of the groups and n is the number of ranked objects (Gibbons, 1976).

Tables 18 through 21 contain the results of this analysis. For each group (White-box and Black-box) a product-moment correlation has been computed to compare with the τ-statistic. The coefficients of concordance show a very strong similarity of rankings among subjects within each group, for LNPR, OPTM, and SCNR (significant at $\alpha = .001$) and a weaker concordance for ITAX ($\alpha = .1$). This suggests that the low-complexity, low-computation ITAX may contain errors of more similar difficulty, thus leading the subjects to generate greater numbers of discovery permutations. Concordance among the total group is high for all programs, which would lead one to anticipate reasonably high correlations between group rank sum. Such is the case with the highest correlation found for the low logical-complexity ITAX and OPTM. This

Table 18

Analysis of Discovery Time Ranks (ITAX)

Error Number	1	1.2	2	3	4	5	6	7	8	9
Mean rank (white-box)	5.77	6.05	5.77	3.64	4.73	5.00	5.77	7.68	4.14	6.45
Ranks	$6\frac{1}{2}$	5	8	1	2	3	4	9	$6\frac{1}{2}$	10
Mean rank (black-box)	6.09	5.36	6.41	3.55	3.77	4.59	5.45	6.45	6.09	6.73
Ranks	6	8	6	1	3	4	6	10	2	9

Notes. Kendall's tau statistic = .586

Significant for $\alpha = .05$

Product-moment correlation = .704

Coefficient of concordance (white-box) = .190
$X^2 = 18.85$ with 10 df

Coefficient of concordance (black-box) = .188
$X^2 = 18.70$ with 10 df

Mean rank (total)	5.93	5.95	6.09	3.59	4.25	4.60	5.61	7.07	5.11	6.59
Ranks	6	7	8	1	2	3	5	10	4	9

Notes. Coefficient of concordance (total) = .161
$X^2 = 32.00$ with 10 df

Table 19

Analysis of Discovery Time Ranks (LNPR)

Error number	1	2	3	4	5	6	7	8	9.1	9.2	10	11	12	13	14.1	14.2
Mean rank (white-box)	5.14	5.41	19.73	7.82	8.32	4.45	10.73	8.23	9.05	9.68	10.73	9.64	9.05	7.50	9.86	9.68
Ranks	2	3	15	5	7	1	14½	6	8½	11½	14½	10	8½	4	13	11½
Mean rank (black-box)	6.09	6.27	9.18	9.36	6.05	3.73	9.18	9.86	9.86	9.00	9.86	8.50	9.86	9.45	9.86	9.86
Ranks	3	4	7½	9	2	1	7½	13½	13½	6	13½	5	13½	10	13½	13½

Notes. Kendall's tau statistic = .272

Significant for $\alpha = .2$

Product-moment correlation = .783

Coefficient of concordance (white-box) = .456 $X^2 = 75.33$ with 10 df

Coefficient of concordance (black-box) = .289 $X^2 = 47.81$ with 10 df

Mean rank (total)	5.61	5.84	9.95	8.59	7.18	4.09	9.95	9.05	9.45	9.34	10.30	9.07	9.45	8.48	9.86	9.77
Ranks	2	3	14½	6	4	1	14½	7	10½	9	16	8	10½	5	13	12

Notes. Coefficient of concordance (total) = .319 $X^2 = 105.59$ with 10 df

Table 20

Analysis of Discovery Time Ranks (OPTM)

Error number	1	2	3	4	5	6	7	8
Mean rank (white-box)	1.95	3.00	2.41	6.36	6.73	4.95	5.59	5.00
Ranks	1	3	2	7	8	4	6	5
Mean rank (black-box)	2.14	2.77	2.18	6.55	6.55	4.77	6.00	5.05
Ranks	1	3	2	7½	7½	4	6	5

Notes. Kendall's tau statistic = .933

Significant for α = .01

Product-moment correlation = .992

Coefficient of concordance (white-box) = .640
X^2 = 49.33 with 10 df

Coefficient of concordance (black-box) = 668
X^2 = 51.50 with 10 df

Mean rank (total)	2.05	2.89	2.30	6.45	6.64	4.86	5.80	5.02
Ranks	1	3	2	7	8	4	6	5

Notes. Coefficient of concordance (total) = .652
X^2 = 100.45 with 10 df

Table 21

Analysis of Discovery Time Ranks (BCNR)

Error number	1	2.1	2.2	2.3	2.4	3	4	5	6.1	6.2
Mean rank (white-box)	18.59	17.41	17.59	18.59	18.59	17.41	17.50	16.64	15.27	18.59
Ranks	27½	19½	22½	27½	27½	19½	21	13	10½	27½
Mean rank (black-box)	18.18	18.18	18.18	18.18	18.18	18.18	15.95	15.95	7.95	15.32
Ranks	25½	25½	25½	25½	25½	25½	11½	11½	2	7

Notes. Kendall's tau statistic = .215

Significant for $\alpha = .01$

Product-moment correlation = .696

Coefficient of concordance (white-box) = .295 X^2 = 97.30 with 10 df

Coefficient of concordance (black-box) = .342 X^2 =113.12 with 10 df

Mean rank (total)	18.39	17.80	17.89	18.39	18.39	17.80	16.73	16.30	11.61	16.95
Ranks	29	23½	26	29	29	23½	16	11½	2	19

Notes. Coefficient of concordance (total) = .270 X^2 = 178.67 with 10 df

Table 21--Continued

Error number	7	8	9	10	11	12	13	14	15	16
Mean rank (white-box)	14.36	17.36	17.59	14.05	5.32	14.05	11.86	18.59	17.05	16.68
Ranks	8	18	22½	6½	1	6½	2	27½	17	14
Mean rank (black-box)	12.82	16.77	15.64	15.45	2.73	13.82	14.14	18.18	18.18	16.77
Ranks	3	16	10	8½	1	4	5	25½	25½	16

Notes. Kendall's tau statistic = .215

Significant for $\alpha = .01$

Product-moment correlation = .696

Coefficient of concordance (white-box) = .295 $X^2 = 97.30$ with 10 df

Coefficient of concordance (black-box) = .342 $X^2 = 113.12$ with 10 df

Mean rank (total)	13.59	17.07	16.61	14.75	4.02	13.92	13.00	18.39	17.61	16.73
Ranks	4	20	14	8	1	6	3	29	21	16

Notes. Coefficient of concordance (total) = .270 $X^2 = 178.67$ with 10 df

Table 21--Continued

Error number	17	18	19	20	21	22	23	24	25	26	27
Mean rank (white-box)	18.59	15.86	12.86	15.27	12.68	14.41	12.14	16.91	18.59	18.59	17.00
Ranks	27½	12	5	10½	4	9	3	15	27½	27½	16
Mean rank (black-box)	18.18	16.86	18.18	18.18	14.91	18.18	16.77	15.45	16.68	17.14	16.68
Ranks	25½	18	25½	25½	6	25½	16	8½	13½	19	13½

Notes. Kendall's tau statistic = .215

Significant for $\alpha = .01$

Product-moment correlation = .696

Coefficient of concordance (white-box) = .295 $X^2 = 97.30$ with 10 df

Coefficient of concordance (black-box) = .342 $X^2 = 113.12$ with 10 df

Mean rank (total)	18.39	16.36	15.52	16.73	13.80	16.30	14.45	16.18	17.64	17.86	16.84
Ranks	29	13	9	16	5	4½	7	10	22	25	18

Notes. Coefficient of concordance (total) = .270 $X^2 = 178.67$ with 10 df

is to be expected for two reasons: First in low complexity programs the number of execution paths is limited and test data will tend to have less of an influence over the stimuli generated by program execution. Second, the number of total errors was small and a much greater percentage of them were found by most subjects. Hence, a massive assignment of mid-ranks for undiscovered errors was not necessary (as it was for SCNR and LNPR), with its concomitant effect of "watering down" rank discrimination.

Simple Correlations

Simple product-moment correlations were computed between all combinations of questionnaire and performance variables (Table 22) as these were needed for a later multiple regression. As will be seen in later sections, many of these variables are subtlely interrelated and simple correlations are insufficient as information for proper interpretation. However, a few observations regarding the strongest relationships will be made. The educational background group were in general positively correlated with all performance measures. The experience category appears to be of mixed value in predicting debugging performance. Of the self-assessment categories, proficiency in the technical areas of computer science principles and algorithms were significantly related to performance, and, to a lesser degree, mathematics and system design. Nearly all variables measuring subject familiarity with the experimental programs were positively correlated with performance. Moreover, familiarity with SCNR (the program which most exemplifies applied computer science) was remarkably effective in predicting overall subject success in the experiment.

Table 22

Simple Correlations of Questionnaire Variables with Performance Measures

	Number of Errors Found						
	Questionnaire Variables	ITAX	LNPR	OPTM	SCNR	Simple Total	Standardized Total
EDUCATIONAL	Age	-.290	+.102	-.031	-.123	-.104	-.119
	Years of Schooling	-.196	+.436	+.326	+.053	+.195	+.215
	Highest Degree	-.070	+.399	+.428	+.212	+.310	+.336
	Computer-Related Course Work	-.104	-.008	+.280	-.103	-.037	+.023
	Computer-Required Course Work	+.076	+.068	+.225	+.109	+.088	+.132
	Theory & Practice Course Work	-.125	+.061	+.049	+.060	+.029	+.015
EXP	Experience General	-.185	+.183	-.147	-.095	-.051	-.085
	Experience Programming	-.121	+.242	-.352	-.104	-.048	-.116
SELF ASSESSMENT	Data Processing Principles	-.133	+.084	-.234	-.101	-.092	-.133
	Computer Science Principles	+.412	+.158	+.015	+.440	+.394	+.355
	Systems Analysis	+.075	+.234	+.207	-.128	+.077	+.134
	Systems Design	+.110	+.388	+.060	+.130	+.252	+.238
	Program Design	+.316	+.170	-.366	+.097	+.137	+.075
	Program Writing	+.143	+.266	-.436	+.222	+.181	+.067
	Program Debugging	+.169	+.333	-.388	+.260	+.240	+.129
	Operations Research	+.111	-.265	+.044	-.042	-.087	-.053
	Probability/ Statistics	+.035	-.023	+.155	-.193	-.073	-.009
	Mathematics	+.235	+.161	+.224	+.078	+.199	+.242
	File Handling	-.006	+.130	-.088	-.077	-.005	-.014
	Algorithms	+.424	+.257	+.280	+.294	+.403	+.435
FAMILIARITY	ITAX-Theory	+.334	+.297	.000	+.103	+.253	+.254
	ITAX-Practice	+.295	+.316	+.088	+.297	+.362	+.345
	LNPR-Theory	-.097	+.064	+.347	-.302	-.098	+.004
	LNPR-Practice	+.120	+.234	+.383	-.005	+.180	+.254
	OPTM-Theory	+.244	+.316	+.307	-.069	+.198	+.276
	OPTM-Practice	+.436	+.398	+.290	+.154	+.390	+.443
	SCNR-Theory	+.182	+.417	+.237	+.625	+.562	+.506
	SCNR-Practice	+.375	+.507	+.434	+.595	+.663	+.662

Multiple Regression Analysis

A series of linear stepwise multiple regression analyses were performed using the performance variables (e.g., number of errors found) as dependent variables and the individual questionnaire data as independent variables. The intent of this analysis was not to arrive at some useful formula for predicting an arbitrary programmer's debugging performance. The small sample size (22) and large number of independent variables (17) employed requires a skeptical attitude toward the reliability of these regression results as some form of screening device for choosing potentially good debuggers. It was hoped, though, that the regression results would confirm some general notions developed by the investigator after many hours of studying the experimental results.

A few observations regarding the subject group composition helps in understanding the results of the regression analysis. The group was specifically chosen for its diversity, as the variability in age, education, and experience indicates. One could also identify subgroups of like subjects: older, more highly educated computing educators, young currently-active system programmers, and middle-aged data-processing oriented applications programmers. There were, in the sample, a selection of extraordinary subjects: a bright sporadically-performing ninth-grade dropout, a very high performing computer science professor with no formal education in the area save self-study, and a highly-educated much-experienced subject with an (apparent) background in systems programming (then education) who performed relatively poorly. The point of these observations is that the "average" group of programmers is quite often similarly diverse and any analysis of a small set of such subjects is apt to reflect the widely-varying personalities of the group as much as the general associations sought.

The variables used in these regression runs included age, educational background, experience, and self-evaluation variables extracted from the questionnaires. A list of the variables used and some mnemonic abbrevations are shown in Table 23 indicating the results of a forward-selection stepwise multiple linear regression using these variables as predictor variables, and the number of errors found by the program as the dependent variable. Table 24 shows the first six variables selected by the regression routine for inclusion in the predictive equation. With each variable is given the sign of the regression coefficient associated with each successive predictor variable.

Performance on ITAX was negatively affected by nearly all educational variables, as well as by familiarity with the theory behind SCNR, normally attained in an academic setting. All other variable inclusions are inexplicable. ITAX was, however, the simplest of the programs and most subjects did fairly well, leading one to conclude that remaining associations (e.g., FPOPTM) may be spurious. Performance on LNPR was expected to be strongly dependent upon an understanding of the problem, an ability to comprehend prolonged (mathematically) algorithmic specification, and experience with some of the subproblems within LNPR (e.g., matrix inversion). Experience with problems like SCNR proved more important, as it proved so in performance on SCNR and OPTM.

Table 23

Mnemonics for Regression Variables

Mnemonic Abbreviation	Variable
AGE	Age
FTSCH	Years of Schooling
HIDEG	Highest Degree
SUCR	Computer-Related Course Work
SURP	Computer-Required Course Work
SUTP	Theory & Practice Course Work
EXPGEN	Experience: General
EXPPGM	Experience: Programming
DPPRIN	Data Processing Principles
CSPRIN	Computer Science Principles
SYSANL	Systems Analysis
SYSDES	Systems Design
PGMDES	Program Design
PGMWRT	Program Writing
PGMDBG	Program Debugging
OPSRES	Operations Research
PROBSTAT	Probability/Statistics
MATH	Mathematics
FILHAN	File Handling
ALG	Algorithms
FTITAX	ITAX-Theory
FPITAX	ITAX-Practice
FTLNPR	LNPR-Theory
FPLNPR	LNPR-Practice
FTOPTM	OPTM-Theory
FPOPTM	OPTM-Practice
FTSCNR	SCNR-Theory
FPSCNR	SCNR-Practice

Table 24

Important Independent Variables in Regression Analysis by Program

	Dependent Variable							
	ITAX Errors		LNPR Errors		OPTM Errors		SCNR Errors	
Order of Entry	Variable	Sign of Coefficient	Variable	Sign of Coefficient	Variable	Sign of Coefficient	Variable	Sign of Coefficient
1	FPOPTM	+	FPSCNR	+	PGMWRT	-	FTSCNR	+
2	FTSCH	-	FTSCH	+	FPSCNR	+	SUCR	-
3	SUCR	-	SUCR	-	DPPRIN	-	HIDEG	+
4	SCPRIN	+	PRGDBG	+	SYSANL	+	FTOPTM	-
5	DPPRIN	-	SYSDES	+	MATH	+	CSPRIN	+
6	SYSANL	+	SUTP	+	FPOPTM	-	MATH	-

Like ITAX most individuals found a fair number of errors in LNPR so that characteristics of the few exceptionally high- and low-performing subjects undoubtedly had a greater effect on the independent variables selected. No reasoning is apparent, however, for the particular choices. Regression results for SCNR are well within expectation. As SCNR was written as a finite state automaton, educational exposure to this formal theory was expected to be important, and proved so with the inclusion of FTSCNR, CSPRIN, and HIDEG. Since language theory, and, for example, compiler courses, are a rather recent academic topic, it is not surprising to find negative regression coefficients for age-related variables such as AGE, SUTP, and SUCR.

The last column of Table 25 shows the results from a regression with a standardized performance measure as the dependent variable. This variable was computed as the sum of the standardized scores on ITAX, LNPR, OPTM, and SCNR. The standardized scores were computed by subtracting from each subject's number-of-errors-found the mean across all subjects, then dividing this difference by the sample standard deviation.

The first variable to enter the regression equation for this standardized total was the ubiquitous FPSCNR. As stated above, it appears that those subjects who had had experience programming applications similar to SCNR performed significantly better than average. Familiarity with parsing algorithms like SCNR would tend to exemplify a sophisticated computer-science-oriented individual with system programming experience, a likely candidate for high performance. Advanced degree attainment appears as an important variable, even though it is

Table 25

Important Independent Variables in Regression Analysis of Total Performance Measures

Order of Entry	Dependent Variables					
	Total Errors Found			Standardized Total		
	Variable	Beta Coefficient	R^2 Increase	Variable	Beta Coefficient	R^2 Increase
1	FPSCNR	+0.34	0.44	FPSCNR	+0.97	0.44
2	SUCR	-2.31	0.19	SUCR	-1.96	0.13
3	HIDEG	+1.15	0.04	HIDEG	+0.81	0.04
4	SURP	+1.92	0.05	FPITAX	-0.06	0.04
5	FPITAX	-0.36	0.05	SURP	+2.00	0.05
6	SYSANL	+0.11	0.02	FTLNPR	+0.94	0.05
7	FTLNPR	+0.94	0.03	SYSANL	+0.28	0.05
8	MATH	-0.75	0.03	MATH	-0.56	0.04

positively correlated with the adversely influencing age measures. Further conclusions and speculations regarding the individual factors affecting debugging performance requires allusion to results obtained outside the regression analysis, and will be postponed until the next chapter.

Analysis of Variance

For each program, the number of errors found by a subject was divided by the length of the debugging sessions (measured in hours), and the total number of errors known to be in the program. This mean percentage of errors per hour (MPEH) measure was employed as the criterion variable in an analysis of variance, performed in conformance with a procedure outlined by Winer (1971). The division by the length of the debugging session normalized the effect of differing time allotments assigned for each program. The division by the known number of errors adjusted the dependent variable (MPEH) for the fact that each of the four programs had a different number of resident errors available for discovery.

The first step in the data analyses is shown in Tables 26 and 27 which reflects the experimental design. Each subject was randomly assigned into one of two groups and each group was assigned four of the eight combinations of program and test data type (see Chapter IV). This design allows within-subject estimates of main-and second-order effects to be made.

Following Winer, the comparisons associated with the effects are computed by calculating the weighted sums of treatment combination

Table 26

Sums of MPEH for Group I

Subject No.	ITAX-BB	LNPR-BB	OPTM-WB	SCNR-WB	Total
1	30	6.25	12.5	2.15	50.9
2	70	14.58	25.0	4.3	113.88
3	40	20.83	25.0	9.68	95.51
4	60	14.58	31.25	9.68	115.51
5	40	10.42	31.25	5.38	87.05
6	50	2.08	31.25	2.15	85.48
7	10	6.25	25.0	3.23	44.48
8	30	4.17	18.75	3.23	56.15
9	20	2.08	18.75	0	40.83
10	80	14.58	37.5	8.6	140.68
11	30	2.08	25.0	3.23	60.31
Total	460	97.9	281.25	51.63	890.78

Table 27

Sums of MPEH for Group II

Subject No.	ITAX-WB	LNPR-WB	OPTM-BB	SCNR-BB	Total
12	50	0	18.75	10.75	79.5
13	40	6.25	18.75	4.3	69.3
14	60	6.25	31.25	9.68	107.18
15	10	4.17	31.25	2.15	47.57
16	50	0	12.5	1.08	63.58
17	30	4.17	37.5	1.08	72.25
18	30	8.33	31.25	7.53	77.11
19	70	8.33	31.25	5.38	114.96
20	40	16.67	31.25	15.05	102.97
21	50	6.25	31.25	2.15	89.65
22	60	2.08	31.25	2.15	95.48
Total	490	62.5	306.25	61.3	920.05

totals, shown in Table 28. These comparisons indicate a strong negative effect of logical complexity and computational content upon error discovery as was to be expected. The table also indicates a large positive interaction effect between these factors. This was also to be anticipated as the MPEH's for LNPR (high LC, high CC) were higher than those for SCNR (high LC, low CC) for many subjects. The summary of this analysis of variance is shown in Table 29. The LC and CC main effects, as well as their interaction, are clearly significant as $F_{.99}(1,61) = 7.08$.

The surprising and counterintuitive result is the positive effect of LC-CC interaction. The subjects found, on the average, more errors within SCNR than within LNPR, but SCNR had almost twice as many known errors, leading to a smaller mean percentage of errors per hour. SCNR was shown in an earlier section to be composed of more difficult errors--those with low visibility. Apparently this difference more than offset the greater number of errors within SCNR. With larger programs like SCNR and LNPR the effects of successively larger error discovery times becomes important. Hence the effect of doubling the number of errors in a program is apparently not reciprocally offset by a proportional increase in error discoveries. This fact may have made SCNR more difficult to debug than LNPR, even though the latter had a higher degree of computational content.

A Distributional Model of Discovery Times

It is reasonable to assume that the search for program errors is indiscriminant, in that, a priori, no error is singled out as the

Table 28

Computation of Treatment Comparisons

Combination	ITAX-BB	LNPR-BB	OPTM-WB	SCNR-WB	ITAX-WB	LNPR-WB	OPTM-BB	SCNR-BB	
Treatment Sums	460	97.9	281.25	51.63	490	62.5	306.25	61.3	Comparison
Overall	+	+	+	+	+	+	+	+	1810.83
Test Type (TT)	+	+	-	-	-	-	+	+	40.07
Log Compl (LC)	-	+	-	+	-	+	-	+	-1264.17
Comp Cont (CC)	-	+	+	-	-	+	+	-	-315.03
TT by LC	-	+	+	-	+	-	-	+	50.07
TT by CC	-	+	-	+	+	-	+	-	80.46
LC by CC	+	+	-	-	+	+	-	-	409.97
TT by LC by CC	+	+	+	+	-	-	-	-	-29.27

Table 29

ANOVA Summary for Experiment

Source	Sum of Squares	Degrees of Freedom	Mean Square	F-Statistic
Between Subjects	3,708.94	21		
TT by CC by LC	21.42	1		
Subjects within groups	3,687.52	20		
Within Subjects	60,628.02	67		
Test Type (TT)	40.14	1	40.14	.178
Log Comp (LC)	39,953.00	1	39,953.00	177.54
Comp Cont (CC)	2,418.00	1	2,418.00	11.03
TT by LC	62.68	1	62.68	.279
TT by CC	161.85	1	161.85	.720
LC by CC	4,201.90	1	4,201.90	18.67
Residual	13,727.41	61	225.04	

object of pursuit prior to its discovery. Hence, one may conclude that the search for all errors proceeds simultaneously in parallel time, and that the differences among discovery times is attributable to the error "obviousness," the programmer's skill in pursuit, and the effects of randomness. The simplest probability model describing the likelihood of error discovery is that of the negative exponential distribution. In adopting it, one assumes that, for an arbitrary error, the probability of finding an error in the next small time quantum is identical irrespective of the length of time devoted thus far to the search.

It was further assumed that the differences between the relative proficiencies of any two subjects could be approximated as a linear effect on the subject's rate of error discovery. The relatively good concordance of error discovery time ranks lends some support to the hypothesis that a subject's skill in debugging is reasonably independent of the particular errors left to discover. In conformance with this assumption, each subject's proficiency was hypothesized to be expressible in the form of a discovery rate multiplier ρ where, for example, ρ_i/ρ_k equals the ratio of subject i's discovery rate to subject j's for an arbitrary error.

A similar assumption was made regarding the relative difficulties of specific errors. Associated with each error j is hypothesized a difficulty index α_j, which equals the basic discovery rate for an error; hence, differences in error difficulty are assumed to be condensible into an arrival rate which is independent of the subject or length of search. Each subject's inherent debugging skill is represented as the subject discovery rate ρ_i. This "skill index" is assumed to be independent of the particular errors being pursued and constant over the debugging interval. Each subject is assumed to be independent of the particular errors being pursued and constant over the debugging interval. Each subject is assumed to have spent some amount of initial time μ_i (that was measured as debugging activity) reading the specifications or performing some other activity during which error discovery was impossible. The associated density function is given by

$$f_{\tilde{t}_{ij}}(t|\alpha_j,\rho_i,\mu_i) = (\alpha_j\rho_i)\, e^{-(\alpha_j\rho_i)(t-\mu_i)} \; .$$

Each error is assumed to be the subject of a search which is independent of the search being conducted simultaneously for all other remaining errors. Furthermore, the activities of each subject are assumed to be independent of all other subjects. Hence, one may assume each error discovery time $\tilde{t}_{ij}$ to be an independent random variable, and express the likelihood of error discoveries and non-discoveries as

$$L(\underline{\alpha},\underline{\rho},\underline{\mu}) = \prod_{1\le i\le M}\left[\prod_{j\in\P_i} f_{\tilde{t}_{ij}}(t_{ij};\alpha_j,\rho_i,\mu_i)\cdot\prod_{j\notin\P_i}[1-F_{\tilde{t}_{ij}}(T)]\right] \qquad (5.1)$$

$$= \prod_{1\le i\le M}\left[\prod_{j\in\P_i}(\alpha_j\rho_i)\, e^{-(\alpha_j\rho_i)(t_{ij}-\mu_i)} \cdot \prod_{j\in\P_i} e^{-(\alpha_j\rho_i)(T-\mu_i)}\right.$$

which may be alternatively expressed as

$$\prod_{1\le j\le N}\left[\prod_{i\in\P_j}(\alpha_j\rho_i)\, e^{-(\alpha_j\rho_i)(t_{ij}-\mu_i)} \prod_{i\in\P_j} e^{-(\alpha_j\rho_i)(T-\mu_i)}\right. \; . \qquad (5.2)$$

For the formulae above and all other discussion of discovery times, the following notations will apply:

- N is defined as the number of errors in the program.
- M is the number of debugging subjects.
- N_i is defined as the number of errors found during debugging by the i'th subject.
- M_j is the number of subjects finding the j'th error.
- t_{ij} is the time to discovery of error number j, by the i'th subject, where j is an arbitrary index over program errors.
- $t_{i(j)}$ is the time to discovery of the j'th error, by the i'th subject, and is an order statistic, where $t_{i(0)} = 0$.
- Δ_{ij} is the time between the discoveries of the j'th and (j-1)'st error, by the i^{th} programmer and $\Delta_{ij} = t_{i(j)} - t_{i(j-1)}$.
- μ_i is a location parameter designating the time at which the i'th subject commenced debugging activity, and $\underline{\mu} = (\mu_1,\cdots,\mu_M)$.
- α_j is a difficulty index, the discovery rate associated with the j'th error, for the "average" subject, and $\underline{\alpha} = (\alpha_1,\cdots,\alpha_N)$.
- ρ_i is a proficiency index, the discovery rate associated with the i'th subject for the "average" error, and $\underline{\rho} = (\rho_1,\cdots,\rho_M)$.
- $\mathcal{I}_j \subseteq \{1,\cdots,M\}$ of cardinality M_j, denotes the index set of subjects who discovered error number j.
- $\mathcal{I}_i \subseteq \{1,\cdots,N\}$ of cardinality N_i, denotes the index set of error numbers found by the i^{th} subject.
- T is the time interval over which debugging was performed.

The first form (5.1) is convenient for deriving the conditions for the maximum likelihood estimator vectors $\hat{\underline{\rho}} = (\hat{\rho}_1,\cdots\hat{\rho}_M)$ and $\hat{\underline{\mu}} = (\hat{\mu}_1,\cdots,\hat{\mu}_M)$. The natural logarithm of the likelihood function is given by

$$\ell n\, L(\underline{\alpha},\underline{\rho},\underline{\mu}) = \sum_{1\le i\le M}\left[\sum_{j\in\mathcal{I}_i}(\ell n\rho_i+\ell n\alpha_j-(\alpha_j\rho_i)(t_{ij}-\mu_i) - \sum_{j\in\mathcal{I}_i}(\alpha_j\rho_i)(T-\mu_i)\right]. \tag{5.3}$$

The likelihood function is an increasing function of μ_i; hence it is maximized at

$$\hat{\mu}_i = \min_j(t_{ij}) = t_{i,(1)} ,$$

the first-order statistic on the set of discovery times for each subject. Maximum likelihood estimators $\hat{\underline{\alpha}}$ and $\hat{\underline{\rho}}$ can be found by classical optimization methods. Let

$$0 = \frac{d\,\ell n\, L(\underline{\alpha},\underline{\rho},\underline{\mu})}{d\,\rho_k} = \left[\sum_{j\in\mathcal{N}_k} \frac{1}{\rho_k} - \alpha_j(t_{kj} - \mu_k) - \sum_{j\notin\mathcal{N}_k} \alpha_j(T - \mu_k)\right]$$

$$= \frac{N_k}{\rho_k} - \sum_{j\in\mathcal{N}_k} \alpha_j\, t_{kj} - \sum_{j\notin\mathcal{N}_k} \alpha_j\, T + \sum_{1\leq j\leq N} \alpha_j\, \mu_k .$$

Then
$$\hat{\rho}_k = \frac{N_k}{\sum\limits_{j\in\mathcal{N}_k} \alpha_j\, t_{kj} + \sum\limits_{j\notin\mathcal{N}_k} T - \alpha_s\, \mu_k} \tag{5.5}$$

for $\alpha_s = \alpha_1 + \cdots + \alpha_N$. The second form of the likelihood function (5.2) is more useful for the derivation of the maximum likelihood estimators for $\underline{\alpha}$. The natural log of this function is given by

$$\ell n\, L(\underline{\alpha},\underline{\rho},\underline{\mu}) = \sum_{1\leq j\leq N}\left[\sum_{i\in\mathcal{N}_j} \ell n(\alpha_j) + \ell n(\rho_i) - (\alpha_j\rho_i)(t_{ij}-\mu_i) \sum_{i\notin\mathcal{N}_j} (\alpha_j\rho_i)(T-\mu_i)\right]$$

and
$$\frac{d\,\ell n\, L(\underline{\alpha},\underline{\rho},\underline{\mu})}{d\alpha_k} = \sum_{i\in\mathcal{N}_k}\left[\frac{1}{\alpha_k} - \rho_i(t_{ik} - \mu_i)\right] - \sum_{i\notin\mathcal{N}_k} \rho_i(T - \mu_i)$$

$$= \frac{M_k}{\alpha_k} - \sum_{i\in\mathcal{N}_k} \rho_i\, t_{ik} - \sum_{i\notin\mathcal{N}_k} \rho_i\, T + \sum_{1\leq i\leq M} \rho_i\mu_i .$$

Then $$\hat{\alpha}_k = \frac{M_k}{\sum_{i\in\mathcal{M}_k} \rho_i\, t_{ik} + \sum_{i\in\mathcal{M}_k} \rho_i + T + \sum_{1\le i\le M} \rho_i\mu_i} \,. \qquad (5.6)$$

In the case that all errors are assumed to be of equal difficulty, then $\alpha_j = \alpha$ for all j, and

$$\hat{\lambda}_i\ M\ \alpha\hat{\rho}_i = \frac{N_i}{\sum_{j\in\mathcal{M}_i} t_{ij} + (N - n_i)T - N\,\mu_i} \qquad (5.7)$$

where $\hat{\underline{\lambda}}$ estimates the arrival rates for each discovery time distribution associated with the M subjects.

The main purpose in hypothesizing the existence of an independent subject skill index (ρ) was that it permitted this investigator to aggregate the discovery times over subjects to obtain more reliable estimates of distributional parameters. A program was written to maximize the likelihood function (5.1) by searching over the parameter space. This program jumped from parameter space point $(\underline{\alpha},\underline{\rho})^{(k)}$ to point $(\underline{\alpha},\underline{\rho})^{(k+1)}$ by a means of a steepest ascent algorithm based upon the Newton-Raphson method. The program converged relatively quickly and consistently to solutions.

This program was run with two settings for the location parameter: $\hat{\mu}_i = t_{i(1)}$ (the first discovery time) and $\hat{\mu}_i = t_{i(1)}/2$. The choice of $\hat{\mu}_i = t_{i(1)}$ was due to its role as the maximum likelihood estimator (MLE) for μ_i. This MLE is always positively biased however, and a more reasonable estimate of subject starting time was sought. Under ideal experimental conditions, μ_i could be assumed to be 0, as subjects would presumably commence with error pursuit immediately. Conversations with some of the subjects, together with comments on the activity logs and unusually large first arrival times, however, made it evident that some nonspecific preparatory time had been spent by subjects during which the discovery of errors was impossible. It was arbitrarily assumed that one-half of the time to each subject's first discovery was uncountable. This assumption is probably in error for those subjects with short first discovery times, but also has little effect on the estimators since it can only differ from the true μ_i by a small amount. Those subjects with long first discovery times may well have spent longer than one-half of this time reading the specifications or some other preparatory activity. By not recognizing this, the errors eventually found will have overstated arrival times and this will tend to decrease the errors' estimated arrival rates. In any event, no better solution seemed evident.

Maximum likelihood estimates $\hat{\underline{\alpha}}$ and $\hat{\underline{\rho}}$ were determined by the search program for each of the four experimental programs. Not surprisingly, the discovery indices for errors ($\hat{\underline{\alpha}}$) corresponded well to the error discovery frequencies. Similarly, the proficiency indices ($\hat{\underline{\rho}}$) were in general conformance with the number of errors found by each subject. For fixed and known $\underline{\mu}$, these estimators are asymptotically unbiased (as N increases). Because $L(\underline{\alpha},\underline{\rho},\underline{\mu})$ is not a concave function, it is not justifiable to use the inverse matrix of second partial derivatives (evaluated at $\hat{\underline{\alpha}},\hat{\underline{\rho}},\hat{\underline{\mu}}$)) as estimates of the sampling covariances among parametric estimators (Bard, 1974). Although these covariances are expected to be quite large, the general relationship of parameters to one another is probably close to that exhibited by the estimates, judging from the way that the search program slowly converged to these estimates without wild fluctuations. The MLE's $(\hat{\underline{\alpha}},\hat{\underline{\rho}})$ found for the likelihood function $L(\underline{\alpha},\underline{\rho},\underline{\mu}=\underline{t}_{(1)}/2)$ are approximations to one of an infinite set of MLE vector pairs. Since every term in (5.1) involving α_j is always multiplied by some ρ_i, for any maximum $(\underline{\alpha},\underline{\rho})$ there exists a maximum $(c\alpha,\rho/c)$, where c is a scalar constant.

One consequence of the nonuniqueness of the MLE values $(\hat{\underline{\alpha}},\hat{\underline{\rho}})$ is that they cannot be compared among different experimental programs. Within any experimental program, however, the ratios of any two subject proficiencies or error difficulties can be computed without concern for the value of c. The ratio of the smallest estimator value to each of the others has been computed for the MLE's found for each of the four experimental programs and is shown in Tables 30 through 33. The range of difficulty index ratios is smallest for ITAX, the simplest of the experimental programs. The remaining programs have a much greater range of difficulty index ratios, indicating the presence of some distinctly nontrivial errors. Like the difficulty index ratios, proficiency index ratios express a difference in error discovery rates. The proficiency index ratios for ITAX, SCNR, and LNPR indicate a reasonably small range of subject debugging rates with the median rate about one-half the rate of the fastest subject's, and twice that of the slowest. Debugging proficiency for OPTM has a much greater range; the results indicate that the fastest subject debugs approximately 20 times faster than the slowest. It should also be remembered that a number of subjects were not included in this analysis for the lack of a sufficient number of discoveries. Hence, it may well be the case

Table 30

Maximum Likelihood Estimators for ITAX
($\hat{\alpha}$, $\hat{\rho}$ with $\mu_i \equiv t_{i(1)}/2$)

Error Number	1.1/1.2	2	3	4	5/6	7	8	9
Difficulty Index ($\hat{\alpha}_j$)	.76	.94	2.42	1.97	1.77	.64	1.5	.48
Ratio of estimate size	1.58	1.96	5.04	4.10	3.69	1.33	3.12	1

Subject Number	1	2	3	4	5	6	7	8	9	10	11	12
Proficiency Index ($\hat{\rho}_i$)	.92	2.58	3.17	1.53	2.23	1.77	1.99	2.28	3.53	1.45	2.54	.81
Ratio of estimate size	1.13	3.18	3.91	1.89	2.75	2.18	2.46	2.81	4.36	1.79	3.13	1

Table 31

Maximum Likelihood Estimators for LNPR
($\hat{\alpha}$, $\hat{\rho}$ with $\mu_1 \equiv t_{1(1)}/2$)

Error Number	1	2	4	5/6	8	9.1	9.2	11	13
Difficulty Index ($\hat{\alpha}_j$)	1.85	2.26	.73	4.17	.26	.24	.25	.46	.61
Ratio of estimate size	7.71	9.42	3.04	17.38	1.08	1	1.04	1.92	2.54

Subject Number	1	2	3	4	5	6	7	8	9	10	11
Proficiency Index ($\hat{\rho}_i$)	1.05	.97	.94	.37	.37	.38	.67	1.19	3.36	.64	.88
Ratio of estimate size	2.84	2.62	2.54	1	1	1.02	1.81	3.22	9.08	1.73	2.38

Table 32

Maximum Likelihood Estimators for OPTM
($\hat{\alpha}$, $\hat{\beta}$ with $\mu_1 = t_{i(1)}/2$)

Error Number	1	2	3	6	7	8
Difficulty Index ($\hat{\alpha}_j$)	4.91	4.70	6.18	.92	.27	.88
Ratio of estimate size	18.18	17.4	22.9	3.41	1	3.26

Subject Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Proficiency Index ($\hat{\beta}_i$)	3.87	1.04	5.61	4.62	5.23	1.3	.8	.64	.79	1.85	1.65	1.94	2.04	.25	2.29	1.09
Ratio of estimate size	15.48	4.16	22.44	18.48	20.92	5.2	3.2	2.56	3.16	7.4	6.6	7.76	8.16	1	9.16	4.36

Table 33

Maximum Likelihood Estimators for SCNR
($\hat{\alpha}$, $\hat{\beta}$ with $\mu_1 = t_{1(1)}/2$)

Error Number	4	5	6.1	6.2	7	9	10	11	12	13	18	19	21	22	23	24	27
Difficulty Index ($\hat{\alpha}_j$)	.29	.27	1.74	.14	1.17	.42	.5	2.93	1.19	1.44	.57	.67	1.27	.48	.41	.34	.36
Ratio of estimate size	2.07	1.93	12.43	1	8.36	3	3.57	20.92	8.5	10.29	4.08	4.78	9.07	3.43	2.93	2.43	2.57

Subject Number	1	2	3	4	5	6	7	8	9	10
Proficiency Index (β_i)	.66	.43	.58	.55	1.01	1.77	.79	.38	1.20	.92
Ratio of estimate size	1.74	1.13	1.53	1.45	2.66	4.66	2.08	1	3.16	2.42

that proficiency ratio ranges of 30 to 1 exist for all four programs. This surprisingly large range in debugging rates agrees with the 27 to 1 range in debugging performance reported by Sackman, Grant, and Erickson (1969) in an early study of this kind.

Test for Exponentiality

The maximum likelihood estimator values found for the model

$$f_{\tilde{t}_{ij}}(t|\alpha_j,\rho_i,\mu_i) = (\alpha_j \; \rho_i) \; e^{-(\alpha_j\rho_i)(t-\mu_i)}$$

were used to transform each t_{ij} to a variable with an identical distribution for all subjects and errors. Using $\hat{\mu}_i = t_{i(1)}/2$ and the MLE's $\underline{\hat{\alpha}}$ and $\underline{\hat{\rho}}$ found by the search routines, each discovery time was converted to a standardized variable z_{ij} with a unit negative exponential distribution. This was accomplished through a change of variables, with

$$\tilde{t}_{ij} = \frac{\tilde{z}_{ij}}{\alpha_j\rho_i} + \mu_i$$

and $$g(z_{ij}) = f(t_{ij}(z_{ij}))\left|\frac{d\ t_{ij}}{d\ z_{ij}}\right| .$$

The resulting distribution of $\tilde{z}_{ij}$ is simply $e^{-z_{ij}}$, and all transformed discovery times are identically distributed, as desired. The frequency distribution of these standardized discovery times is shown in Figure 14.

It is difficult to determine the fit of these data to a negative exponential model, as a great number of errors went undiscovered across the ensemble of subjects. Hence, to the right of the histogram

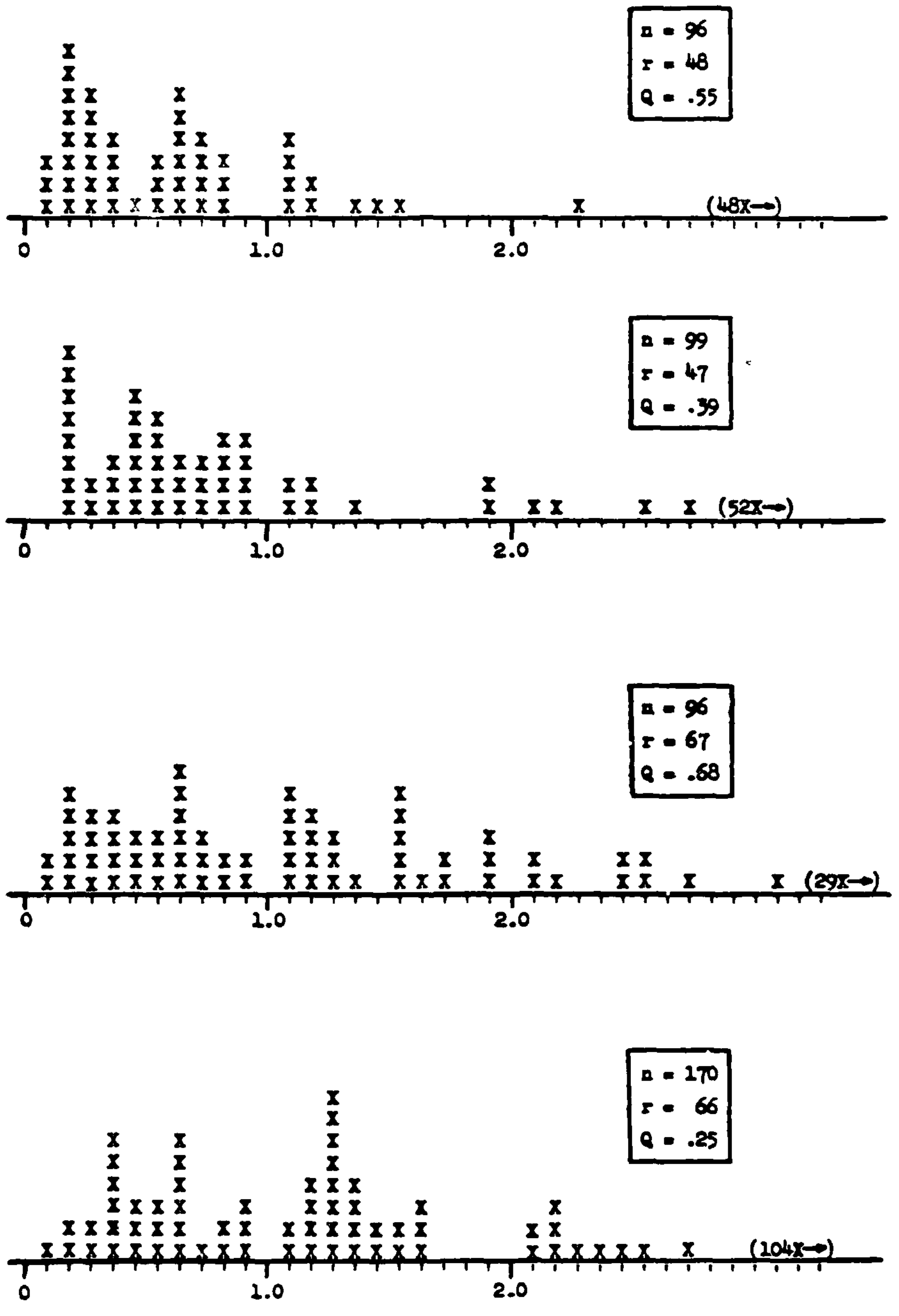


Figure 14. Sample distributions of standardized discovery times

scale in the figure reside a number of "eventual" discovery times, the exact values of which could not have been obtained without extending the experimental sessions indefinitely. However, a test for conformance to a negative exponential distribution does exist for censored samples such as this.

The Gnedenko test is among the most powerful tests of exponentiality for censored samples (Monn, Schaefer, & Singpurwalla, 1974). The test statistic, Q, is derived by computing

$$Q(r_1,r_2) = \frac{\sum_{i=1}^{r_1} (S_i/r_1)}{\sum_{i=r_1+1}^{r} (S_i/r_2)}$$

where each S_i is a weighted interarrival time of the form

$$S_i = (n - i + 1)(t_{(i)} - t_{(i-1)}) \qquad (t_{(0)} = 0)$$

and n is the number of sample points. The terms r_1 and r_2 indicate an arbitrary partitioning of the r available arrival times, where n-r data points are unavailable (censored). $Q(r_1,r_2)$ is approximately F-distributed with $2r_1$ and $2r_2$ degrees of freedom. Because the $r(=r_1+r_2)$ ordered arrival times can be split anywhere for calculation of Q, an equal split $(r_1=r_2)$ is often employed. When $r_1=r_2$ the expected value for Q is 1, under the assumption that the arrival times are identically distributed, negative exponential random variables.

For each set of standardized discovery times, a Q value was calculated. With samples of the sizes obtained from the experiment, any Q value less than 2/3 or greater than 3/2 indicates a significant departure from exponentiality. As shown in Figure 14 every set of discovery +imes generated a Q value substantially less than one. This result is normally accepted as evidence of a decreasing hazard rate (the instantaneous discovery rate). Even after normalizing each error's discovery time by its difficulty index, it appears that error discoveries become decreasingly probable as time goes on. Further speculations regarding the cause of this phenomenon are given in the conclusions.

Software Reliability Models

The simplest and most-used model of software failure behavior is that proposed by Jelinski and Moranda (1973) in which software failure interarrival times are assumed to conform to a stepwise decreasing negative exponential distribution. This was shown in Chapter III to be equivalent to a model in which all program errors are identically distributed, negative exponential random variables with common mean time to failure and arrival rate. The preceding analyses in this chapter have shown the error arrival rates to be significantly different within a program, even after accounting for subject differences. This finding is in agreement with Littlewood (1979) who demonstrated that the Jelinski-Moranda model underestimated the mean time to failure in many cases. Littlewood also showed that the Jelinski-Moranda model dictated a distribution of number of errors remaining (by time t) which was insufficiently skewed in the right tail when compared to real data obtained from large programming projects.

These observations by Littlewood apply equally well to the discovery times obtained from this experiment. The maximum likelihood estimates for difficulty indices varied substantially from error to error. If errors have a similar range of difficulty in all programs, one would expect the number of errors found by time t distribution to be significantly more skewed than if all errors were equally difficult. Moreover, even when error discovery times were standardized by the differing difficulty indices, the Gnedenko test indicated (significantly) the effect of a decreasing discovery rate over time. This finding further demonstrated the failings of the Jelinski-Moranda model for the experimental data obtained in this study.

To demonstrate the inaccuracy of the stepwise decreasing exponential model, the discovery times for all subjects were aggregated (for each program) by computing discovery rates $\hat{\lambda}_i$ as shown in equation (5.7) in the preceding section. These discovery rates were used to create standardized discovery-time variables

$$z_{ij} = (t_{ij} - \hat{\mu}_i)\hat{\lambda}_i$$

which normalized for the effect of different subject proficiencies and starting times. In place of N in (5.7), a maximum likelihood estimate $\hat{N}$ was used.

The development for these MLE's is straightforward. Each subject found n_i errors and failed to discover $N-n_i$ at a rate $\lambda_i = \alpha\rho_i$ where α is the common error difficulty index and ρ_i the assumed subject proficiency index. Discovery times t_{ij} are reduced by μ_i equalling one-half the first discovery time for each subject. Then the likelihood of any one subject's particular error discoveries (including the order in which they were found) is given by

$$\prod_{j \in \P_i} \lambda_i e^{-\lambda_i t_{ij}} \cdot (e^{-\lambda_i T})^{N-N_i}$$

Errors are assumed to be indistinguishable, and so the joint likelihood function for the ordered sample, including all subjects is given by

$$L = \left(\frac{N!}{(N-n)!}\right) \prod_{1 \leq i \leq M} \left[\lambda_i^{N_i} \prod_{j \in \P_i} e^{-\lambda_i t_{ij}} \cdot e^{-\lambda_i T(N-N_i)}\right]$$

where $$n = \sum_{1 \leq i \leq M} N_i \ .$$

The MLE for each subject's discovery rate is derived as

$$\ln L = \ln N! - \ln(N-n)! + \sum_{1 \leq i \leq M} \left[N_i \ln \lambda_i - \sum_{j \in \ _i} \lambda_i t_{ij} - \lambda_i T(N-N_i)\right]$$

and $$\frac{d \ln L}{d\lambda_k} = \frac{N_k}{\lambda_k} - \sum_{j \in \P_k} t_{kj} - T(N-N_k)$$

implying that $$\hat{\lambda}_k = \frac{N_k}{\sum\limits_{j \in \P_k} t_{kj} + T(N-N_k)} \ .$$

An estimator $\hat{N}$ for total program errors is determined by calculating

$$\frac{d \ln L}{dN} = \sum_{0 \leq i \leq k-1} \frac{1}{(N-i)} - T \sum \lambda_i = 0$$

and finding the root $\hat{N}$ of this equation in N. The estimates $\hat{N}$ were computed for each of the four experimental programs. The true values for N were available, of course, and equalled the number of known errors in a program times the number of subjects for whom discovery times were analyzed. The values for N and $\hat{N}$ are given in Table 34. The predicted number of resident errors exceeded the known number greatly for all programs. The reason for this is that the estimating routines "made up" for the declining rate of error discovery by predicting that a larger number of errors were being found at a fixed rate.

Evaluation of the Order Statistic-Based Model

In Chapter III, a software reliability model was introduced, in which error discoveries (or software failures) were assumed to be order statistics on an underlying mixture distribution. Under the assumption of exponential discovery times and gamma-distributed arrival rates, each error discovery was found to be distributed as

$$f(t;\alpha,\beta) = \frac{\alpha^{\beta}\beta}{(t+\alpha)^{\beta+1}} ,$$

the Pareto distribution discussed by Littlewood (1978). The ordered discovery times obtained from the experiment were used to compute the maximum likelihood estimates for $\hat{\alpha}$, $\hat{\beta}$, and $\hat{n}$ for the likelihood function

$$f_{\tilde{t}_{(1)},\ldots,\tilde{t}_{(k)}}(Z_{(1)},\ldots,Z_{(k)}) = L(\beta,\beta,n) =$$

$$\frac{n!}{(n-k)!} \; \frac{\alpha^{n\beta}\beta^{k}}{(Z_{(k)}+\alpha)^{\beta(n-k)} \prod_{1\le i\le k} (Z_{(i)}+\alpha)^{\beta+1}} .$$

Table 34

Comparison of True and Estimated
Error Population Size

Program	True N	MLE N
ITAX	96	221
LNPR	99	304
OPTM	96	184
SCNR	170	743

For each experimental program the ordered observation vector. $Z_{(*)}$, was composed of all standardized discovery times recorded for all subjects in the reduced discovery time set (see Tables 14 through 17). A standardized time for the i'th subject and error j was calculated as

$$Z_{ij} = \hat{\rho}_i(t_{ij} - \hat{\mu}_i)$$

by using the sets of maximum likelihood estimates $\hat{\rho}_i$ and $\hat{\mu}_i$ given in Tables 29 to 32. It was assumed that this normalization would adjust for individual subject's starting times and discovery rates, and yield standardized times for aggregation. The set of times $\{z_{ij}\}$ was then ordered to produce the order statistic vector $Z_{(*)}$.

A program was written to conduct a Newton-Raphson search over the parameter space $\{(\alpha,\beta,n)\}$ for constant values given by $Z_{(*)}$ and k, and with the objective function

$$\ell n\, L(\alpha,\beta,n) = \ell n(n!) - \ell n(n-k)! + n\beta\, \ell n(\alpha) + k\, \ell n(\beta) -$$

$$\beta(n-k)\ell n(Z_{(k)} + \alpha) - \sum_{1\le i\le k} (\beta+1)(Z_{(i)} + \alpha) .$$

As can be seen from the log-likelihood function, for fixed n, increases in α could be paired with decreases in β and result in similar objective function values. Unlike the $(\hat{\alpha},\hat{\rho})$ MLE search for equation (5.1), the reciprocal nature of (α,β) pairs resulted in prolonged searches over the $\{(\alpha,\beta,n)\}$ space. This iterative search yielded log-likelihood values differing by a fraction of a percent for vastly differing MLE triples.

The optimal MLE values found by the search program were uniformally discouraging. In every case, the MLE $\hat{n}$ coverged upon a value slightly greater than k, the number of known error discoveries as shown in Table 35.

Table 35

Comparison of True and Estimated Error Population Size

Program	k	True n	Estimated $\hat{n}$
ITAX	48	96	52
LNPR	47	99	56
OPTM	67	96	77
SCNR	67	170	76

The search program was re-executed to search over $\{(\alpha,\beta)\}$ fixing n at the true value for each experimental program; the resulting log-likelihood values differed very little from the optimal values.

This evidence was sufficient to convince us that the Gamma-Exponential model was ill-suited for describing discovery time behavior. Like Littlewood (1978) we could have used successive MLE's to determine if successive ordered discovery times produced uniformly distributed fractiles when applied to the next-discovery-time distribution. This was not done for two reasons. First, the (unconditional) discovery time distribution was proved non-exponential by the Gnedenko test (in contradiction to our model), and the gamma distribution for difficulty indices was primarily chosen for its flexibility and mixing compatibility with the exponential (as opposed to having chosen it for some form of representativeness). Second, the proper estimation of error population size seems to us an important property of a software reliability distribution, and this property should be a necessary property of an adopted model.

CHAPTER VI

FINDINGS, CONCLUSIONS, AND SUGGESTIONS FOR FURTHER RESEARCH

Findings

An exploratory study was conducted to determine what personal, environmental, and program-related factors affect the process of discovering errors in computer software. In a controlled setting, a group of software professionals searched for errors in programs whose error composition was known, and which had been developed to represent various degrees of computational content and logical complexity. The subject group exhibited a wide range of ages, experience, and education, whereas less diversity existed among subjects' self-assessment and prior exposure to the program applications. Subject performance was measured by the number of errors each found and this performance measure was positively correlated with both general and computer science-related education. Other analyses indicated that the most important factor in performance was the amount of recent programming experience held by a subject. Subject performance also differed from program to program, and an analysis of variance led to the singling out of logical complexity as the most significant factor, with the degree of computational content a distant (but significant) second and test data type exhibiting no significant effect on subject performance. A significant negative interaction effect between logical complexity and computational content signified deficiency in the measurement of debugging performance.

An analysis of the errors found by subjects indicated that logical and data-handling errors were more difficult than computational errors, and that the total set of errors represented a wide range of associated difficulties. There existed a similarly wide range in subject proficiencies. Maximum likelihood estimates of error difficulties and subject proficiencies were obtained and a test for exponentiality of normalized discovery times indicated a significant departure from exponentiality, due to the presence of a decreasing instantaneous discovery rate over time. To this decreasing discovery rate can be attributed the subsequent overestimation of error population size by the Jelinski-Moranda and our proposed software reliability models.

Problem Background

The program error has served this study as the single subject of research and experimentation. Yet, ten years ago this study would have been impossible to conduct properly, because of the crucial lack of a conceptual framework with which view the programming process. During the first thirty years of the computer era, the program error was often viewed as some form of retribution to those who were not properly trained or sufficiently creative software artists. During the 1970s, however, the program error has risen in stature. Methodologies have evolved that would help limit its numbers, pursue its incarnations and estimate the number of survivors.

This interest in error prevention, detection, and correction has evidenced itself in functional computer research areas: Operating systems must not only allocate resources but also protect the entire software system from the damage that may result from errors hiding in one system element; programming languages must now be more than comprehensive, powerful, and efficient--they must help express intent and facilitate proofs of program correctness; hardware must exhibit fault-tolerance as well as fault detection.

Alongside the growth of these traditional areas of computer study have evolved disciplines addressing directly the problems of software "mis-construction." These areas may be loosely aggregated as the field of validation research, and further bifurcated into the realms of verification and testing. Verification research is largely a product of the computer science community and clearly the more "optimistic" of the two approaches. The most important goal of verification research is the development of conceptual and material tools with which to prove program correctness before, during, and after its development. The long-range aim of verification research is the development of a language with which to express problems and the identification of correctness-preserving transformations that guarantee the absence of errors. Until that end is achieved there is the clear need for methodologies which aid in error discovery.

These methodologies include the area of software testing which [illegible] gained increased acceptance as an important research area as well [illegible] The questions addressed within testing [illegible] minimal complete test sets and the procedures for generating flowchart-covering input data. Of primary concern is the notion of test criterion reliability: the demonstration that a procedure for generating test data will find any errors in a program.

The most immediate and obvious response by the computing community to the crisis in software reliability has been through the expansion and redefinition of the testing phase of software development. Most of what is now testing was once considered an integral part of program writing--a part that should not have been necessary, but (somehow) always was. An enlightened and mature software development community is now beginning to consider testing in the same positive way that product assurance is conducted in other manufacturing processes. The painful experiences of the past have forced software professionals to develop testing innovations like automated test tools, requirements-based testing procedures, and independent testing departments.

Between the rigorous expositions of formal testing theorists and the published recommendation of testing practitioners lies the body of literature regarded as Software Engineering. The title of software engineer indicates an expression of professionalism and acceptance of responsibility beyond that of the "sensitive software artist." Like other engineering disciplines, the software engineer seeks to develop methods and procedures which bring precision into the construction of software architectures.

The foremost interest among Software Engineers is Reliable Software. It is the theme of conferences and the topic of journals.

Having reliable software as a goal is the admission of the inevitability of systemic imperfection. A Reliable Software product (as opposed to a correct one) is a system which performs what is expected of it in a satisfactory manner over a reasonable period of time. The reliable system is attained by gathering information about its behavior and correcting deviations in this behavior from that intended. Viewed in this fashion, the minimization of errors becomes the dual of the problem of maximizing reliability.

Software Reliability Theorists model the attainment of program reliability as a process obeying general principles, irrespective of the software being tested, the testing approach used, or the individuals involved. The search for errors in the time domain is viewed as a stochastic process with error discovery as the significant random event. The overriding motivation for this viewpoint is <u>prediction</u>. If the reliability of a software system is a function of time, then the determination of that function permits the software engineer to predict the time necessary to achieve a given level of reliability, the estimated number of remaining errors, and other indicators of reliability growth. In the real world of software development, these predictions must be made anyway, since budgets must be set and manpower planned. Whether software reliability theory is yet mature enough to merit the attention of software developers is the subject of debate. The fact remains, however, that in the absence of models of program behavior and reliability growth, the software developer is left only with intuition and speculation.

Experimental Approach

This study was initiated so that a better understanding of software validation might be gained through the comparison of current approaches and the analysis of experimental results. It was apparent from discussions with leading practitioners, and researchers in the field, that at least two distinct approaches were available to the researcher seeking to test models of the error discovery process. The deterministic view is held by most computer scientists and practioners. The success of a software development project is seen as a function of the development environment and the nature of the project in question. This approach to understanding the "software problem" focuses upon the influence of tools, personnel, and test methods upon the quality of the software product. The second approach is tied to the stochastic model of software reliability theorists in which any software good is viewed as a belonging to a group whose members conform to approximate model of aggregate behavior. Both of these approaches are important in the same way that micro- and macroeconomics participate in describing economic behavior.

An experiment was designed so that data could be collected and analyzed using both approaches for understanding program errors and their discovery. From the many variables differentiating software testing situations, three "objects" were chosen as the most important for analysis. First, it was decided that the variability among people may account for many of the differences in software testing success. A group of 22 subjects was selected to participate in a debugging experiment in which some relationships might be discovered between subject background and characteristics, and aspects of their performance. These subjects were professionals in the computing field, and intentionally chosen to represent a variety of experiential and academic backgrounds.

The second source of experimental variability was the set of programs given to the subjects for debugging. Degree of logical complexity (LC) and computational content (CC) were chosen as the two program characteristics that might have the greatest effect on one's ability to find errors. An objective measure of each characteristic was decided upon and one program was written to represent each of the f our combinations of factor settings: low LC, low CC (ITAX); low LC, high CC (OPTM); high LC, low CC (SCNR); high LC, high CC (LNPR). Each program was thoroughly debugged and each error found was documented, then categorized. These errors were reinserted into the experimental programs which were then given to the subjects for testing and error correction.

The third controlled experimental factor was the type of input data given to each subject for the purpose of generating test results. One type of test data was generated by "black-box" methods--the development of data which checks the conformance of program processing to the specifications. In addition to these black-box data, each program's control graph was determined and, from an anlysis of execution paths, sets of "white-box" data sets were developed. These two test generation procedures correspond to the two extremes between which resides the proper balanced approach of combining functional testing with testing based upon program structure.

Conclusions

Although many conclusions can be drawn from the relationships made apparent during this research, the evidence supporting these conclusions was as much the result of ongoing subjective analysis as it was the explicit testing of preformulated hypotheses. As an exploratory study such things, for example, as the relationship between program complexity and failure time distributions were completely obscure before conducting the experiment and by no means transparent now. We will attempt to segregate those conclusions drawn from objective analysis from those whose underpinnings are based more upon opinion. It is hoped, in this way, that future researchers may concentrate their energies in the retest and interpretation of the latter realm of observations.

The novelty in this study rests in the choice of true replicates for the observations of debugging performance. To our knowledge, only Howden (1978) and Hetzel (1975) have employed identical programs as the instrument stimulating observations which can be compared. Howden, however, was the sole subject of his own experiment, employing various test criteria (as mechanically as possible) on a series of programs to judge the efficacy of these criteria. Hetzel's experiment was most similar to ours, and in fact serves as the model for many of the procedures which we adopted and some which we avoided. Hetzel wrote three programs, reinserted the errors, and had a group of subjects conduct a timed search for evidence of these errors.

This evidence could be "conclusive" that is, the discovery of improper program statements, or "indirect"--the observation of improper output. Moreover, Hetzel's subjects were not permitted to modify the experimental programs, and were timed indiscriminately during all phases of program debugging. Lastly, Hetzel's subject sample was reasonably homogeneous--young programmers and graduate students with backgrounds in the computer sciences.

We endeavored to inject some further precision into this format, while broadening the range of experimental objects so that the results could be more generally applied. The added precision resulted from a conscious, objective search for programs which differed from one another in measurable ways, and the partitioning of debugging activity into the subactivities of code/results review, error correction, and test data construction. Many of the subjects complained good naturedly that the experiment had seemed like going to work, except that it had been a longer day. This was exactly the result that was desired. The subjects could work at their own pace with few constraints on the manner with which they normally found and corrected errors.

The subject set was remarkably diverse in background . The range of organizations from which the subjects were recruited and the subject selection criteria were formulated to help ensure a broad sample. The older subjects typically had less training in computer studies and many had acquired their knowledge and proficiencies on the job. The younger subjects tended to have more formal education in computing and a better familiarity with some of the algorithmic models employed in SCNR and LNPR.

The most obvious question of interest regarding the outcomes is "Who found the most errors and why?" Five of the top seven performers were in their late twenties, all but one had done master's work in a tech al field, and all had four or more continual years of recent programming experience. We conclude that to maintain proficiency in debugging it is necessary to continue its practice. In itself, this is not surprising, as atrophy of other technical skills is a well-known phenomenon. It is surprising however that <u>ceteris parabis</u> this aspect was more important than familiarity with the problem, or formal training in computer studies. Prior training in a technical area was also predominant among the top performers. It is, however, difficult to draw conclusions from this, however, as it is impossible to determine if these academic fields simply have appeal to those with programming aptitude or engender some reasoning powers in the student that later evidences itself in programming proficiency.

Subject age, and variables positively correlated with it, frequently entered the regression analysis on performance and nearly always with a negative coefficient. The simple correlation coefficient of age and number of errors found was highly negative and only three of the nine subjects over thirty found more than the median number of errors. Nevertheless, we do not believe age in itself to be an important factor in predicting debugging performance. Many of the older subjects had only moderate <u>recent</u> experience in day-to-day programming, and had academic training that either excluded computer studies altogether, or consisted of computer classes of questionable merit in the distant past.

The quality of experience also appeared to have an influence on debugging performance. Those subjects whose normal programming involved system software were uniformly above the median. Conversely, subjects whose programming duties revolved around normal data processing (the applications of conventional businesses) found far fewer errors, and the more conventional were their day-to-day programming tasks (e.g., programming and maintenance of accounting systems), the worse was their performance. One reason for this finding may be the rather esoteric nature of three of the experimental programs, LNPR, SCRN, and OPTM. However, many of the highest performing subjects expressed no prior familiarity with one or more of the applications. Moreover, the five best performers overall also scored highest on ITAX, a reasonably conventional application.

Educational background was measured in a variety of ways, including highest degree attained and number of units earned. In terms of simple correlations with total performance, it appeared that more schooling led to a greater number of errors found. In many of the regression analyses, however, the only educational variable to enter with a positive coefficient was highest degree earned. Because many of the subjects were still going to school, educational attainment measures were highly confounded with age and experience measures, as well as with one another. Moreover, the quality of the educational experiences measured was decidedly varied within the sample. Aggregated together were part-time and full-time enrollments, Ivy League and training school educations, a mid-sixties art major, and a late-seventies computer science graduate. Any further research attempting to relate education to programming performance should consider that even the five educational measures employed in this study were not sufficiently discriminating.

Both experience in general and experience in programming exhibited mildly negative correlations with error discovery, although neither variable ever entered the regression analysis for any of the experimental programs or for total performance. Because a moderate amount of recent experience seemed to be a deciding factor in subject performance, the young subjects with little experience and older subjects with much past experience effected insignificant regression results by the confounding of age and experience. It should be pointed out, however, that the top five performers each had professional experience in programming that was well above the median.

Familiarity with the environmental aspects of the experiment was measured by asking each subject his/her past experience with the BASIC language and interactive systems. The response to these questions appeared to have no bearing on subject performance. This is not particularly surprising as the BASIC subset and interactive system chosen were relatively unsophisticated and easy to use. There apparently exists no tertiary relationships which relate normal programming environment of the subject and performance in this experiment. One reason for this may be that the subject had the opportunity to debug in a manner with which he/she felt most comfortable. Some subjects used the computing system in a "batch-like" fashion; studying the listing for some time while accumulating and recording errors found, then making corrections _en masse_ to produce a new run and listing.

Others employed what one subject termed the "run-and-gun" method only practicable on interactive systems. This approach entailed frequent interactions with the computing system by applying program changes as soon as possible, some exploratory, some corrective. This difference may reduce to the differences in the subjects' need for stimuli, attention span, or discipline, and would serve as the basis of an interesting study of its own.

Prior to the experiment each subject was asked to evaluate himself/herself in 12 programming-related areas. The subjects were asked to be as objective as possible and were given no reference with which to make comparison. From the repeated assurances that were demanded by the subject for anonymity, it was felt that these assessments were reasonably honest evaluations on the part of the subjects. A simple sum of ranks was computed for each subject for an analysis that led to surprising results. The sample range for these total self-assessment scores was very small, and a simple correlation with total performance was nearly zero. The top six performers' total self-assessment scores were sprinkled uniformly throughout the range.

Individual self-assessment categories exhibited slightly stronger relationships to performance, with "knowledge of computer science principles" and "knowledge of algorithms" showing significant positive correlations with error discovery. In the regression analyses, only "knowledge of computer science principles" and "knowledge of data processing principles" entered the regression equation more than once, the former with positive coefficients, the latter with negative coefficients. This appears reasonable as many of the top performers had a more technical background, whereas many of the poor performers were engaged in more mundane data processing work.

We find the lack of discriminating power of these self-assessments to be quite remarkable. When given no reference group with which to compare themselves, subjects on the whole tended to rate themselves uniformly above average. For insight into the reason for these homogeneous assessments, one need only study one of the self-assessment categories. In the mathematics category, for example, some subjects with sound technical backgrounds and some with two years of business school education rated themselves as average. A similarly mixed set of individuals rated themselves as strong, including a system programmer with a master's degree in mathematics and a systems analyst with a degree in public administration. One explanation for these results is that one's standards of excellence increases as one acquired education and experience. Another explanation may be that an unreferenced self-assessment, like that in this study, tends to measure self-confidence, a trait that is not necessarily correlated with education, experience, or proficiency.

In contrast to the self-assessment categories, questions regarding subjects' prior experience with problems similar to the experimental programs appeared more closely related to performance. Responses to these familiarity categories, however, were not necessarily well correlated with the programs indicated in the questions, with the exception of SCNR. Instead, these indications of prior exposure to the theory and application of each experimental program were, as a group, related to all performance measures. It would appear, therefore, that questions regarding prior exposure are more discriminating than generalized self-assessment, and decidedly important in all debugging environments.

Notwithstanding the reasonably small size of the experimental programs, it is felt that conclusions regarding debugging performance can be generalized to the components of medium-and large-scale software. The composition of error classes within the experimental programs was considerably similar to those reported in other studies of medium- and large-scale software (see Chapter II). Subject success in finding errors of each class was also in conformance with that commonly proposed in the literature. The most interesting result of this section of the error analysis was the evidence that error discovery is closely related to the visibility of program misconstruction. This was no doubt in part due to the availability of explicit specifications with which subjects could "pattern-match" the code. Further research along these lines could, however, provide a sounder basis for explaining why certain error types are more readily found than others, which may, in turn, lead to beneficial influences on future language design and test tools.

An analysis of variance was performed to determine the effects of logical complexity, computational content, and test data type upon debugging performance. The overwhelming effect of logical complexity and moderate effect of computational content upon error discovery was predictable and readily observable by studying the raw data. The significantly negative interaction effect of logical complexity and computational content was the most interesting finding of this analysis, for it signified a failure in measurement, or linearity assumptions, or both.

The reader may recall that logical complexity and computational content were measured objectively by the application of formulae to the programs' code (see Appendix B). Possible inaccuracies in these metrics led us to dichotomize the measure range into high and low regions, by which each factor setting was determined. The logical complexity metric value for SCNR (14.8) was, however, quite a bit larger than that for LNPR (10.7), as was the computational content value for OPTM (84%) was greater than that of LNPR (74%). A more accurate assessment of these factors' effects may have been obtained by a multiple regression analysis employing metric values directly and a dummy variable for test data type, in place of our analysis of variance.

The criterion variable for the analysis of variance was the mean percentage of errors found per hour (MPEH). The comparability of MPEH among differing programs is suspect, as it is not clear that dividing the number of program errors found by the number of errors resident and by the debugging duration leads to comparably normalized measures of performance. A better (and more costly) experiment could be devised wherein the same number of errors would be reinserted into the experimental programs, and the same length of time would be given for debugging. Any nonlinear effects of duration and error population sizes could be mitigated, and linear models of factor effects could be more confidently applied.

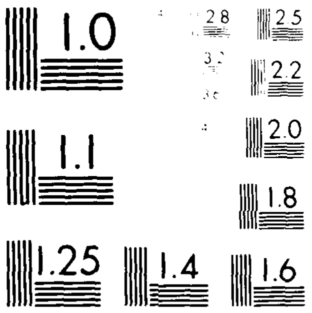
The most basic result shown to be of overwhelming generality was the inequality of expected error discovery times. It has always

AD-A110 366 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT O--ETC F/G 9/2
A STUDY OF FACTORS AFFECTING SOFTWARE TESTING PERFORMANCE AND C--ETC(U)
JUN 80 J L BAHR

UNCLASSIFIED NL

3 OF 3
AD A
110366

END
DATE
FILMED
03-82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4
2.0
1.1
1.8
1.25
1.4
1.6

been quite clear that some errors are discovered much later than others, but never conclusively demonstrated that the order of discovery times was not the result of a random process. Replicating over debugging session was sufficient to show, by inspection, that certain errors were clearly more obscure. A chi-square analysis bore out this inequality of expected discovery times, as did the vastly differing maximum likelihood estimator values for error "difficulty indexes."

A more specific finding was evident from the analysis performed upon the ranks assigned, by subject, in order of error discovery. Not only were certain errors discovered more generally early or later, but the order of all error discoveries was similar for all subjects. This conclusion was based upon the rejection of a random configuration hypothesis by use of Kendall Coefficient of Concordance, and makes it clear that the equal mean-time-to-failure assumption underlying the Jelinski-Moranda model is not justified.

A surprising result also reported in the rank analysis was the high correlations between the two sets of discovery order rank means associated with groups using black-box and white-box data. Combined with the fact that the analysis of variance on error discovery counts showed the groups to be similar, one can only conclude either that theoretical differences between these approaches are not borne out in reality, or that the programs were not large enough systems that the test sets represented radically different input subdomain. We suspect that the answer lies somewhere between.

The greatest benefit of obtaining replications of error discovery time sets for the same programs, was the ability to test software reliability models defined in terms of known, arbitrarily indexed errors. In the absence of replicated discovery time ensembles, it is natural to define failure models in terms of successive inter-arrival times. For most published failure models, however, this procedure makes the hidden assumption that any error is equally likely to occur (which coincidently dispells any concern with order statistics).

To employ the replications of discovery times, it was necessary to find a means of aggregating the performances of quite dissimilar subjects. The simplest means of doing so was the hypothesis that each subject's error discovery rate was proportional to any other subject, for <u>all errors</u>. Hence, the discovery rate of a particular subject for a particular error was assumed to be the product of an individual's proficiency index ρ_i and the error's inherent arrival rate α_j. The maximum likelihood estimates for the set of proficiency indices $\underline{\rho}$ and error difficulty indices $\underline{\alpha}$ were obtained and used to transform all discovery times for each program to normalized values adjusted for individual and error differences. Since many errors went undiscovered, a distributional test (Gnedenko's) for exponentiality was used, which employed censored samples. Even after adjusting for vastly differing error difficulty indices, the normalized discovery time sets exhibited a decreasing rate of discovery.

This result contradicts the assumptions of the Jelinski-Moranda and Schick-Wolverton models, the former assuming a constant discovery rate and the latter hypothesizing an increasing discovery rate for <u>unnormalized</u> discovery times. If anything, the short debugging durations allotted and learning effects of the subjects would tend to increase the observed discovery rates over time, and yet a statistically significant dropoff in these rates was indicated by Gnedenko's test. The only conclusion that can be drawn from this finding is that the longer an error goes undiscovered, the less is the likelihood of its discovery in the near future. This unfortunate effect is more pronounced when the possibility of varying error difficulties is introduced, as the more obscure errors will tend to be found latest in the debugging task and at a slower rate than earlier, more lucid errors.

There is evidence in the literature to support this conjecture of decreasing error discovery rates. Schick and Wolverton (1978) report on a set of error data for which the best distributional fit was obtained by employing a Weibull distribution with a decreasing discovery rate parameter. Littlewood (1978) analyzed a large set of software failure times and found that the exponential model was not sufficiently skewed to the right, indicating increasingly greater overestimation of failure probability as time goes on. Littlewood's solution was to keep the assumption of exponential arrival times and impose the assumption that successive error's failure rates were random variables distributed by related gamma distributions.

We proposed a software reliability model somewhat related to that of Littlewood's. Error difficulty indices were assumed to be gamma distributed and serving as the parameter for exponentially distributed discovery times (not interarrival times like that assumed by Littlewood). From our order statistic based model we derived maximum likelihood estimates of the number of resident errors, and these were much larger than the number of known errors. We believe that it is the assumption of exponentiality that underlies the failure of our model in this regard, as well as that of Jelinski-Moranda and Schick-Wolverton. We further believe that the adequate distributional fits reported by Littlewood (1978) are the result of the exceedingly flexible model which he employs and we expect to find after further research that his method is as inaccurate in predicting the number of resident errors as it is unrepresentative of the phenomenon that he is attempting to model.

We now believe more strongly than before in the strength of generalized order statistic-based software reliability model. The conformance of error discovery rank orders across subjects leads us to conclude that errors are indeed individuals and that probability models must be formulated for actual errors and not associated with the order in which errors are observed (as implicitly assumed with models which employ interarrival times). The rejection of the assumption of discovery time exponentiality causes the theoretical collapse of the Littlewood and Jelinski-Moranda models, as exponentiality of discovery times is a necessary condition for exponentiality of interarrival times. Our model need only be adjusted by finding the proper distributions for errors' shape parameters (which clearly differ from error to error), and the discovery or arrival time distribution best suited to employ these (random) parameters. It is quite possible that the flexibility in discovery rate afforded by the Weibull distribution may serve this purpose. Our own further research will proceed along these lines, and these arguments are offered to the rest of the software reliability community as suggestions for further study.

In summary, the findings of this study have shown that the quality of program testing and debugging is substantially affected by the proficiency of the individual engaged in this activity and characteristics of the software under test.

The most prolific error discoverers were generally well-educated with technical backgrounds. The range of proficiency was broad, with the best subject finding errors at 20-30 times the rate of the worst. Neither the subject's usual programming language nor his normal computing mode (batch or interactive) had any effect on performance. Moreover, the subjects' own assessment of their skills and proficiencies had no bearing on the types or number of errors that were found. Subjects with recent experience in sophisticated applications generally performed best on all the experimental programs and there is every reason to believe that this finding carries over into professional environments.

The strongly adverse effect that logical complexity was found to exert on program debugging effectiveness suggests that the application of structured programming and use of block-structured languages can help improve software testing activities, in addition to their other acknowledged benefits (Meyers, 1976).

The type of test data employed in debugging had little effect on subject performance in this study, although the particular errors found differed from one test data type to the other. Hence, we suggest that both "black-box" and "white-box" methods be employed in practice to maximize testing coverage. We are joined in this opinion by Meyers (1979).

One of the most interesting findings of this study was the decided nonhomogeneity of the error population for each program. A hierarchy among errors based upon their apparent difficulty, was confirmed by subject after subject, as evidenced by the concordance among the subjects' orderings of error discoveries. On the basis of this finding, we conclude that the remainder of any debugging effort is time-consuming not only because there are fewer errors to find, but also because those errors remaining are inherently more subtle. This supposition contradicts the assumptions of the Jelinski-Moranda software reliability model which assumes that the software-failure rate during test is inversely proportional to the number of remaining errors. In fact, after adjusting error discovery times for differences in error difficulties we found that the discovery rate still decreased over time. This conclusion is in accord with the debugging experiences on the practicing programming community and better explains the propensity for software projects to conclude the testing activity over budget.

We conclude that all but the most trivial programs are composed of errors whose mean-times-to-discovery differ substantially. This being the case, the time necessary to find the first error, the last error, or any error between is an order statistic whose distribution is based upon the underlying distributions of the individual errors. We believe that our own attempt to fit an order statistic-based model failed due to the decreasing discovery rate exhibited by the experimental data, even after adjusting for varying difficulties. It is quite possible that the gamma distribution was inappropriate for error difficulty indices, and it was proven that the exponential distribution was unrepresentative of discovery times. We suspect that these distributions can be modelled as a family whose members differ parametrically. The findings of our study convince us that the determination of the proper underlying distributions for discovery times and their distributional parameters is possible through experimentation.

BIBLIOGRAPHY

BIBLIOGRAPHY

Alberts, D. S. The economics of software quality assurance. Proceedings of the 1976 National Computer Conference. New York: AFIPS Press, 1976. Pp. 230-238.

Ashcroft, E., & Manna, Z. The translation of GOTO programs to WHILE programs. Computer Science Report Number CS-188 (Stanford University), 1970.

Baker, F. T. Structured programming in a production environment. IEEE transactions on software engineering. New York: IEEE Press, June 1975. Pp. 241-252.

Bard, Y. Nonlinear parametric estimation. New York: Academic Press, 1974.

Basin, S. L. Estimation of software error rates via capture-recapture sampling. Palo Alto, Calif.: Science Applications, September 1973.

Bell, T. E., & Thayer, T. A. Software requirements: Are they really a problem. Proceedings of the Second International Conference on Software Engineering. New York: IEEE Press, 1976. Pp. 61-68.

Berge, C. Graphs and hypergraphs. Amsterdam: North-Holland, 1973.

Boehm, B. W. Software and its impact: A quantitative assessment. Datamation, May 1973 19(5), 48-59.

Boehm, B. W., McClean, R. K., & Urfrig, D. B. Some experience with automated aids to the design of large-scale reliable software. IEEE transactions on software engineering. New York: IEEE Press, March 1975. Pp. 125-133.

Boyer, R. S., Elspas, B., & Levitt, K. N. SELECT: A formal system for testing and debugging programs by symbolic executions. Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE Press, 1975. Pp. 234-245.

Brooks, F. P. The mythical man-month. Reading, Mass.: Addison-Wesley, 1972.

Brown, A. R., & Sampson, W. A. Program debugging. London: MacDonald, 1973.

Brown, J., & Lipow, M. Testing for software reliability. Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE Press, 1975. Pp. 518-527.

Brown, J. R. Why tools? Proceedings of the Eighth Annual Symposium on Computer Science and Statistics, UCLA, Los Angeles, February 12-13, 1975, pp. 34-42.

Caine, S. H., & Kent, E. PDL--A tool for software design. Proceedings of the 1975 National Computer Conference. New York: AFIPS, 1975. Pp. 314-319.

Christofides, N. Graph theory--An algorithmic approach. New York: Academic Press, 1975.

Clarke, L. A system to generate test data and symbolically execute programs. (CS-060-75). Boulder: University of Colorado, Department of Computer Science, February 1975.

Clarke, L. Generating test data and symbolically executing programs written in ANSI FORTRAN. IEEE transactions on software engineering. New York: IEEE Press, September 1976. Pp. 196-207.

Clarke, L. A. Testing: Achievements and frustrations. Proceedings of the Second International Computer and Applications Conference. New York: IEEE Press, November 1978. Pp. 310-314.

The computer industry. Waltham, Mass.: International Data Corporation, 1978.

Cornell, L., & Halstead, M. H. Predicting the number of bugs expected in a program module (CSD-TR/205). Lafayette, Ind.: Purdue University, October 1976.

Dahl, O-J., Dijkstra, E. W., & Hoare, C. A. R. Structured programming. London: Academic Press, 1972.

David, H. A. Order statistics. New York: Wiley, 1970.

Davis, C. G., & Vick, C. R. The software development system. IEEE transactions on software engineering. New York: IEEE Press, January 1977. Pp. 69-84.

Dorn, P. H. 1979 budget survey. Datamation, January 1979, 25(1), 67-89.

Elspas, B. A comparison of formal verification, symbolic execution, and formal testing. Unpublished note, SRI, June 1977.

Elspas, B., Levitt, K. N., & Waldinger, R. J. An assessment of the techniques for proving program correctness. Computing Surveys, June 1972, 4(2), 13-34.

Fosdick, L., & Osterweil, L. J. Data flow analysis in software reliability. Computing Surveys, September 1976, 8(3), 305-330.

Fujii, M. Independent verification of highly reliable programs. Proceedings of the First International Computer and Applications Conference. New York: IEEE Press, 1978. Pp. 38-44.

Funami, Y., & Halstead, M. H. A software physics analysis of Akiyama's debugging data (CSD-TR/44). Lafayette, Ind.: Purdue University, May 1975.

Gabow, H. N., Maheshwari, S. N., & Osterweil, L. J. On two problems in the generation of program test paths. IEEE transactions on software engineering. New York: IEEE Press, September 1976. Pp. 227-231.

Gerhart, S. L., & Yelowitz, L. Observations of fallibility in the application of modern programming methodologies. IEEE transactions on software engineering. New York: IEEE Press, September 1976. Pp. 195-207.

Gibbons, J. New parametric methods for quantitative analysis. New York: Holt, Rinehart & Winston, 1976.

Goel, A. L., & Okumoto, K. Bayesian software prediction models, volume I: An imperfect debugging model for reliability and other quantitative measures of software systems (RADC-TR-78-155). New York: Rome Air Development Center, 1978.

Goodenough, J. B., & Gerhart, S. L. Toward a theory of test data selection. IEEE transactions on software engineering. New York: IEEE Press, June 1975. Pp. 156-173.

Green, T. F., Schneidewind, N. F., Howard, G. T., & Pariseau, R. Program structures, complexity, and error characteristics. Proceedings of the Symposium on Computer Software Engineering. New York: Polytechnic Press, 1976. Pp. 139-154.

Haberman, A. N. Path expressions (Tech. report). Pittsburgh, Pa.: Carnegie-Mellon University, 1975.

Halstead, M. H. Elements of software science. New York: Elsevier, 1977.

Harary, F. Graph theory. Reading, Mass.: Addison-Wesley, 1972.

Hetzel, W. C. Program test methods. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

Hetzel, W. C. Comparing and analyzing techniques for detecting errors in computer programs. Unpublished doctoral dissertation, Duke University, 1975.

Hoffman, R. H. Automated verification system user's guide. TRW note, 72-FMT-891, 1972.

Hoffman, R. H. User information for interactive automated test data generator (ATDG) system. NASA Johnson Space Center, internal note 75-FM-88, January 1976.

Howden, W. E. Reliability of the path analysis testing strategy. IEEE transactions on software engineering. New York: IEEE Press, September 1976. Pp. 208-215.

Howden, W. E. Symbolic testing and the DISSECT symbolic evaluation system. IEEE transactions on software engineering. New York: IEEE Press, July 1977. Pp. 339-349.

Howden, W. H. Empirical studies of software validation. Tutorial: Software testing and validation techniques. IEEE Computer Society, 1978. Pp. 280-285.

Huang, J. C. An approach to program testing. Computer Surveys, September 1975, 7(3), 113-128.

Jelinsky, Z., & Moranda, P. B. Applications of a probability-based model to a code reading experiment. Proceedings of the IEEE Symposium on Computer Software Reliability. New York: IEEE Press, 1973. Pp. 78-81.

Kennedy, J. E. A survey of automated computer program verification tools. Aerospace Corporation, August 15, 1974.

King, J. A new approach to program testing. Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE Press, 1975. Pp. 228-233.

King, J. Symbolic execution and program testing. Communications of the ACM, July 1976, 18(7), 23-30.

Knuth, D. E. An empirical study of FORTRAN programs (TR CS-186). Palo Alto: Stanford University, 1971.

Krause, K. W., Smith, R. W., & Goodwin, M. A. Optimal software test planning through automated network analysis. Proceedings of the IEEE Symposium on Computer Software Reliability. New York: IEEE Press, 1973. Pp. 18-22.

Littlewood, B. Theories of software reliability: How good are they and can they be improved? Unpublished manuscript, February 1979.

Littlewood, B., & Verrall, J. A Bayesian reliability growth model for computer software. Applied Statistics: Journal of the Royal Statistical Society, Series C, 1973, 22(3), 212-231.

Love, L. T., & Bowman, A. B. An independent test of the theory of software physics. ACM SIGPLAN Notices, November 1976, pp. 4-9.

McCabe, J. T. A complexity measure. IEEE transactions on software engineering. New York: IEEE Press, December 1976. Pp. 308-320.

Meyers, G. The art of software testing. New York: Wiley, 1979.

Miller, E. F., & Melton, R. A. Automated generation of testcase datasets. Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE Press, 1975. Pp. 51-58.

Miller, E. F., Paige, M. R., & Benson, J. P. Structural techniques of program validation. Digest of Papers: 13th IEEE Computer Society International Conference. New York: IEEE Press, 1976. Pp. 31-40.

Mills, H. D. On the statistical validation of computer programs. IBM FSD unpublished paper, July 1970.

Mills, H. D. Mathematical foundations for structured programming. IBM Corporation, Federal Systems Division, FSC 71-5108, 1971.

Monn, R., Schaeffer, N., & Singpurwalls, M. Reliability theory and statistics. New York: Holt, Rinehart & Winston, 1977.

Moranda, P. B. Prediction of software reliability during debugging. Proceedings of the 1975 Annual Reliability and Maintainability Symposium, 1975, Los Angeles, September 18-20, pp. 27-30.

Moranda, P. B. Limits to program testing with random numbers inputs. Proceedings of the Second International Computer and Applications Conference. New York: IEEE Press, 1978. Pp. 521-526.

Musa, J. D. A theory of software reliability and its application. IEEE transactions on software engineering. New York: IEEE Press, September 1975. Pp. 212-230.

Nelson, E. C. A statistical basis for software reliability assessment. TRW Software Series, TRW-SS-73-03, March 1973.

Osterweil, L. J., & Fosdick, L. D. DAVE--A validation error detection and documentation system for FORTRAN programs. Software Practice and Experience, 1976, 6, 35-52.

Paige, M. R. Program graphs, an algebra, and their implication for programming. IEEE transactions on software engineering. New York: IEEE Press, September 1975. Pp. 286-291.

Phister, M. Data processing technology and economics. Santa Monica, Calif.: Santa Monica Publishing, 1976.

Ramamoorthy, C. V. Design and construction of an automated software evaluation system. Proceedings of the 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE Press, April-May 1973. Pp. 28-37.

Ramamoorthy, C. V., & Ho, S. F. Testing large software with automated software evaluation tools. IEEE transactions on software engineering. New York: IEEE Press, March 1975. Pp. 46-58.

Ramamoorthy, C. V., Ho, S. F., & Chen, W. T. On the automated generation of program test data. IEEE transactions on software engineering. New York: IEEE Press, December 1976. Pp. 293-300.

Ramamoorthy, C. V., Kim, K. H., & Chen, W. T. Optimal placement of software monitors aiding systematic testing. IEEE transactions on software engineering. New York: IEEE Press, December 1975, Pp. 403-410.

Reifer, D. J., & Trattner, S. A glossary of software tools and techniques. Computer, July 1977, 18(2), 121-131.

Ross, D. T. Guest editorial--reflections on requirements. IEEE transactions on software engineering. New York: IEEE Press, January 1977. Pp. 2-5. (a)

Ross, D. T. Structured analysis: A language for communicating ideas. IEEE transactions on software engineering. New York: IEEE Press, January 1977. Pp. 16-33. (b)

Ross, D. T., & Schoman, K. E. Structured analysis for requirements definition. IEEE transactions on software engineering. New York: IEEE Press, January 1977. Pp. 26-30.

Rubey, R., Dana, J., & Riche, P. Quantitative aspects of software validation. IEEE transactions on software engineering. New York: IEEE Press, June 1975. Pp. 26-30; 150-155.

Schick, G. S., & Wolverton, R. W. Assessment of software reliability. TRW Software Series, TRW-SS-73-04, September 1972.

Schick, G. S., & Wolverton, R. W. An analysis of competing software reliability models. IEEE transactions on software engineering. New York: IEEE Press, March 1978. Pp. 104-120.

Shooman, M. L., Schick, G. S., & Wolverton, R. W. Types, distribution, and test and correction times for programming errors. IEEE transactions on software engineering. New York: IEEE Press, March 1975. Pp. 11-18.

Stay, J. F. HIPO and interactive program design. IBM Systems Journal, September 1976, pp. 1-17.

Stevens, W. P., Myers, G. J., & Constantine, L. L. Structured design. IBM Systems Journal, March 1974, pp. 12-27.

Stucki, L. G. New directions in automated tools for improving software quality. In Current trends in programming methodology. Englewood Cliffs, N.J.: Prentice-Hall, 1977.

Sullivan, J. E. Measuring the complexity of computer software. Bedford, Mass.: Mitre Corporation, June 1973.

Teichroew, D., & Hershey, E. A. III. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems. IEEE transactions on software engineering. New York: IEEE Press, January 1977. Pp. 18-29.

Thayer, T. A., Lipow, M., & Nelson, E. C. Software reliability study (TRW Software Series, TRW-SS-76-03). Redondo Beach, Calif.: TRW, March 1976.

Walker, M. G. A theory for reliable software. Datamation, September 1978, pp. 12-13.

Walston, P., & Felix, J. Error distributions in program codes. Proceedings of the Second International Computer and Applications Conference. New York: IEEE Press, 1978. Pp. 123-126.

Wegner, P. Research directions in software technology. Proceedings of the Third International Conference on Software Engineering. New York: IEEE Press, 1978. Pp. 243-259.

Zelkowitz, M. V. Perspectives on software engineering. ACM Computing Surveys, June 1978, 10(2), 1-13.

APPENDICES

APPENDIX A

QUESTIONNAIRE INSTRUCTIONS

QUESTIONNAIRE INSTRUCTIONS

1. NAME
2. AGE
3. SEX
4. GENERAL EDUCATIONAL BACKGROUND

List, from earliest educational experience to most recent, the periods in your life in which you were enrolled in some form of schooling. Examples are given below for each column of the table you are requested to fill in.

Educational Environment	Examples include High School, Undergraduate, Training School (CDI, etc.), Graduate School
Period of Enrollment	Examples include Sept. 1975-June 1978, academic years of 1973 to 1975
Major or Emphasis	Examples include Music, Data Processing, Computer Science, None (if applicable)
Title of Degree	Examples include B.S., MBA, PhD, None (if applicable)

5. EDUCATIONAL BACKGROUND IN COMPUTER-RELATED STUDIES

List the number of units earned in courses directly related to studies of computer systems, programming, theory, or applications.

State the number of units in terms of quarter units or semester units.

Estimate, for each period of enrollment, the percentage of your course work which required significant amounts of programming (say, one writing one large program or 5 to 10 smaller ones).

6. EDUCATIONAL BACKGROUND IN PROGRAMMING THEORY AND PRACTICE

List the number of units earned in courses dealing directly with topics in programming theory and practice.

Examples of such courses include language courses (COBOL, FORTRAN, BASIC), Algorithms, Data Structures, Programming Principles.

Courses such as Systems Analysis, Numerical Analysis, Computing Theory or Management Information Systems do not qualify.

Include any courses that you have taught but not formally taken.

7. PROFESSIONAL EXPERIENCE INFORMATION SCIENCES AND DATA PROCESSING

List all distinct experiences that you have had in the data processing and computer science field, not required by courses that you have taken. These experiences may include jobs, consulting engagements, contract work, independent development of skills (such as computing-as-a-hobby), or the donation of your skills.

For each separate experience, state a short description or job title, the duration of the experience in man-months, the time period over which this experience spanned, and the percentage of these man-months devoted to program writing and debugging.

For a part-time job of, say, 10 hours a week in which ½ of the work was system design and ½ programming over a period from June 1, 1977-August 1, 1978, an entry in the table would appear as

Job Title	Job Duration	Time Period	% of Time Programming
⋮	⋮	⋮	⋮
Programmer/Analyst (part-time)	14/4 = 3.5 man-months	to Aug. 1, 1978	50%

8. PROGRAMMING LANGUAGE EXPERIENCE

Of all the programming required of you in educational environments (see question 5), estimate the percentage of the time you used each of the programming languages listed in the table provided for this question. Reply by filling in the table under the column titled EDUCATIONAL.

Of the programming required of you during your professional experiences estimate the percentage of the time you used each of the programming languages listed in the table provided for this question. Reply by filling in the table under the column titled PROFESSIONAL.

9. BATCH/INTERACTIVE EXPERIENCE

We define a BATCH environment as a program development and test environment in which

- programs are coded and keyed onto cards
- the programmer compiles and/or runs the program to obtain a hard copy listing and/or program output
- the program is reviewed and debugged by inspecting the listing for some length of time
- modifications are made to the program card deck to reflect desired program changes
- the program deck is resubmitted to begin the cycle again

We define an INTERACTIVE environment as a program development and test environment in which

- programs are entered into a terminal
- programs are run at the terminal to obtain program results
- results are studied at the terminal
- the program is listed at the terminal and edited to reflect desired program changes
- the cycle is re-begun by running the program

What percentage of your programming experience was acquired in situations more closely resembling a batch environment, and what percentage was acquired in situations more closely resembling an interactive environment?

10. SELF ESTIMATION

How would you estimate your abilities, proficiency, and knowledge in each of the following areas?

- data processing principles
- computer sciences principles
- systems analysis
- program design
- program writing
- program debugging
- operations research
- statistics and probability
- mathematics
- file handling
- algorithms

QUESTIONNAIRE

1. NAME: ______________________________

2. AGE: __________

3. SEX: Male___ Female___

4. GENERAL EDUCATIONAL BACKGROUND

Educational Environment	Period of Enrollment	Major or Emphasis	Title of Degree

5. EDUCATIONAL BACKGROUND IN COMPUTER-RELATED STUDIES

Educational Environment	Period of Enrollment	Semester Units	or Quarter Units	% Requiring Programming

6. EDUCATIONAL BACKGROUND IN PROGRAMMING THEORY AND PRACTICE

Educational Environment	Period of Enrollment	Semester Units	or Quarter Units

7. PROFESSIONAL EXPERIENCE IN INFORMATION SCIENCES AND DATA PROCESSING

Job Title	Job Duration	Time Period	% of Time Programming

8. LANGUAGE EXPERIENCE

	EDUCATIONAL	PROFESSIONAL
ALGOL		
Assembler Language		
BASIC		
COBOL		
FORTRAN		
JOVIAL		
PASCAL		
PL/I		
Total	100%	100%

9. BATCH/INTERACTIVE EXPERIENCE

	Batch	Interactive	TOTAL
Experience Breakdown			100%

10. <u>SELF ESTIMATION</u>

	VERY WEAK	WEAK	AVERAGE	STRONG	VERY STRONG
DATA PROCESSING PRINCIPLES					
COMPUTER SCIENCE PRINCIPLES					
SYSTEMS ANALYSIS					
SYSTEMS DESIGN					
PROGRAM DESIGN					
PROGRAM WRITING					
PROGRAM DEBUGGING					
OPERATIONS RESEARCH					
PROBABILITY AND STATISTICS					
MATHEMATICS					
FILE HANDLING					
ALGORITHMS					

NAME: ____________________

QUESTIONNAIRE II

Please devote some time to read the program specifications included in the packet. I am sure that you will find the experiment much more enjoyable if you have prepared by studying these specifications to determine what it is the programs are supposed to do.

After reading and studying the program specifications, please indicate in the boxes below, the extent of your familiarity with the "theory" behind these applications, as well as the amount of prior experience you have had in programming problems of these kinds.

Now that you have read and studied the program specifications for each of the programs (SCNR, OPTM, LNPR, and ITAX)...

(1) How familiar are you with the "theory" used to solve these problems? Your familiarity with the principles underlying these programs may have come from your schooling, reading, or research on the job. Please answer below under THEORY.

(2) What is the extent of your experience in designing, programming, and/or debugging programs which dealt with problems like those addressed by the experimental programs? Your experience may have been acquired in school, on the job, or through recreational computing. Please answer below under PRACTICE.

	THEORY					PRACTICE				
	Very Familiar	Familiar	Vaguely Familiar	Not Familiar with This Particular Problem	Not Familiar with Any Problem Like This	Much Experience with This Particular Problem	Moderate Experience with This Particular Problem	A Little Experience with This Particular Problem	A Little Experience with Problems Like This	No Experience with Problems Like This
SCNR										
OPTM										
LNPR										
ITAX										

APPENDIX B

PROGRAM METRICS

COMPUTATIONAL CONTENT

The degree of computational content was measured by an "operator ratio" motivated by Halstead's approach (1977) to decomposing programs into indivisible tokens. Each basic construct in a programming language is categorized as one of two token types: operator or operands. An operand is any object of an action and includes data varibles and constants. Operators are program symbols which have zero or more operands as functional arguments, and effect program execution.

Each indivisible construct in the BASIC language was assigned membership into the operator or operand class. Some constructs, though always appearing together, were counted as two or more operators if one of the constructs could appear separately as an operator. String and numeric variables, IMAGE strings and unvarying data for tables were counted as operands. Operators were dichotomized into two classes: computational and noncomputational. Computational operators are those program tokens which take one or more data as operands and produce a result. Noncomputational operators are all others which affect the order of program execution (control), provide input or output (I/O), or provide definition (declarative). A list of the specific operators recognized and a designation of category is given in Table B.1.

A correction factor suggested by Halstead was applied to operator and operand counts. Halstead reasons that any common sub-expression used repeatedly represents a value to a programmer and is not as computationally complex as its constituent operators may imply. With this in mind, the operator and operator counts were reduced by those tokens associated with common expressions in close textual proximity to one another.

The "operator ratio" was calculated as the ratio of the number of computational operators to the total number of operators. It was assumed that the higher this ratio becomes, the greater is the degree of "data processing" per program symbol. Operator ratios were calculated for a dozen programs from the Hewlett Packard Contributed Library,

Table B.1

Operator Classes for BASIC

Construct	Type	Number of Operators	Example
Dimension	Noncomputational	1	DIM ...
Loop	Noncomputational	1	FOR ... = ... TO ... NEXT ...
Input	Noncomputational	1	READ ....
Output	Noncomputational	1 2 2	PRINT expression list PRINT expression list ; PRINT USING ....
Conditional-transfer	Noncomputational	1	IF...THEN....
Unconditional-transfer	Noncomputational	1	GOTO ...
Computed-GOTO	Noncomputational	1	GOTO ... OF ...
Subroutine-call	Noncomputational	1	GOSUB ....
Subroutine-return	Noncomputational	1	RETURN
Arithmetic-operator	Computational	1	+, -, /, *, ↑
Grouping operators	Computational	1	(...)
Indexing operators	Computational	1	[...]
Assignment operator	Computational	1	=
Relational operator	Computational	1	<,>, <=,>, =, = , ≠

chosen to represent a variety of assumed degrees of computation. The operator ratios computed for this sample ranged from 55% to 85% with most in the 65% to 75% interval.

On this basis programs with ratios of 65% or less were designated as having low computational content and those with 75% or more were designated as having high computational content.

The four programs eventually employed as the experimental programs were judged as the best experimental instruments on the basis of their operator ratios, as well as other factors. The operator ratios for these programs are displayed in Table B.2.

Table B.2

Operator Ratios for the Experimental Programs

Program	Number of Computational Operators	Number of Noncomputational Operators	Total	Ratio	Designation
ITAX	166	83	249	67%	LOW
LNPR	322	112	434	74%	HIGH
OPTM	162	30	192	84%	HIGH
SCNR	254	158	412	62%	LOW

As can be seen from Table B.2, ITAX and OPTM had higher ratios than the presumed ideals for membership in their respective classes, while LNPR and SCNR had lower than ideal ratios.

This phenomenon occurred repeatedly during the selection of the experimental programs and can be attributed to the effect of program length on the ratios. During the comparison of Library programs and development of programs as experimental candidates, it became evident that higher operator ratios became more difficult to find as programs of greater and greater length were observed. On this basis, the resulting experimental programs were deemed sufficient as representative of their respective computational content classes, even though differing slightly within like classes.

LOGICAL COMPLEXITY

Two measures were employed in the assessment of logical complexity for experimental program candidates. The simpler of these two was McCabe's complexity measure (1976) corresponding to the cyclomatic number of a program's control graph or flowchart. In practice, this can be calculated as $e - n + 1$, where n equals the number of blocks in a flowchart and e the number of control flow arcs.

A measure developed by TRW (1973) is much more sophisticated and takes into account direction of program branching as well as the degree of nesting at the point of conditional transfer. TRW's Logical Complexity Metric is computed as:

$$L/S + C_1 + C_2 + B/1000$$

where

L = number of logical statements (GOTO, GOSUB, IF, FOR-NEXT)

S = number of executable statements

C_1 = measure of loop complexity

C_2 = measure of IF statement complexity

B = number of separate paths from any segment to any other.

C_1 and C_2 are weighted sums of the number of conditional branches and loops (respectively) at each level of nesting.

The complexity metric values calculated for each experimental program is given below in Table B.3:

Table B.3

Complexity Metrics for the Experimental Programs

Program	McCabe's Metric	Metric	Designated Logical Complexity
ITAX	15	1.6	LOW
LNPR	39	10.7	HIGH
OPTM	3	0.65	LOW
SCNR	56	14.8	HIGH

Like the operator ratio, the complexity metrics appeared to be sensitive to program length--longer programs invariably resulted in greater metric values. Since, however, logical complexity is normally believed to increase with the size of the task, this property of the metrics did not appear inappropriate.

APPENDIX C

EXPERIMENTAL PROGRAM SPECIFICATIONS AND LISTINGS

ITAX

Program Overview

The ITAX program computes the federal and state income tax for residents of a fictitious country.

Program Inputs

Data should be entered beginning on line 9900 in the following order:

9900 DATA W_0, W_1, A_0, R, Y_0, Y_1, Y_2, G_0, G_1
9910 DATA C_0, C_1, C_2, C_3, C_4, C_5, D_0, D_1, D_2
9920 DATA D_3, D_4, T_0, T_1, T_2, T_3, A_1, E

Program Outputs

The program outputs a table showing

- Income Sources
- Income Offsets
- Net Taxable Income
- Income tax
- Withholding
- Tax due

An example follows:

```
                         State    Federal
 Income Sources    =    23178  =   22920  =
 Income Offsets    =     5937  =    5835  =
 Net Taxable       =    17233  =   17085  =
 Income Tax        =      345  =    1709  =
 Withholding       =     1500  =    4000  =
 Tax Due           =    -1155  =   -2291  =
```

<u>Program Processing</u>

Taxes are payable on an amount equal to the total of sources minus the total of offsets. <u>Total sources</u> is equal to the sum of

(state and federal)	1.	Alimony received
(federal only)	2.	State refund
(state and federal)	3.	Salary and wages
(state and federal)	4.	Any excess of dividends received over $100.
(state and federal)	5.	Any excess of gambling winnings over losses.
(state and federal)	6.	One-half of the excess of long-term capital gains over losses and carryover.
(state and federal)	7.	One-half of the excess of short-term capital gains over losses and carryover.
(state and federal)	8.	Interest earned

<u>Total offsets</u> is equal to the sum of

(state and federal)	1.	The lesser of $3000 and the excess of long-term losses and carryover over long-term gains.
(state and federal)	2.	The excess of a + b + c over 3% of earned income
		a. the lesser of hospitalization cost $2000
		b. the excess of drugs and medicine expense over ½% of earned income
		c. ½ of medical insurance cost
(state and federal)	3.	Casualty Loss
(state and federal)	4.	The lesser of charitable deductions and 25% of earned income
(state and federal)	5.	Real and Personal Tax
(federal only)	6.	State Income tax paid
(federal only)	7.	State gas tax
(state only)	8.	Federal gas tax
(state and federal)	9.	Alimony paid
(state and federal)	10.	Employee Business expense

The rates of tax on taxable income are given below as well as the definitions of program variables:

Taxes Payable/Refundable	
Federal	P_0
State	P_1
Withholding	
Federal	W_0
State	W_1
Alimony Received	A_0
State Refund Received	R
Earned Income	
Salary and Wages	Y_0
Dividends	Y_1
Interest Earned	Y_2
Gambling	
Winnings	G_0
Losses	G_1
Capital Transactions	
Long Term	
Gains	C_0
Losses	C_1
Carryover	C2
Short Term	
Gains	C_3
Losses	C_4
Carryover	C_5
Deductions	
Health	
Drugs and Medicine	D_0
Hospitalization	D_1
Health Insurance	D_2
Casualty Loss	D_3
Charitable	D_4

Taxes	
Real and Personal	T_0
State Income	T_1
State Gas	T_2
Federal Gas	T_3
Alimony Paid	A_1
Employee Business Expense	E

Federal and state tax are computed on a progressive scale. Higher brackets of marginal income are taxed at higher rates. The table below shows the <u>marginal</u> rate at which income earned in that bracket is taxed:

Income	Federal Marginal Rate	State Marginal Rate
FIRST $20,000	10%	2%
NEXT $15,000	20%	4%
NEXT $10,000	30%	7%
NEXT $5,000	40%	12%
ANY AMOUNT OVER $50,000	50%	18%

```
20   DIM X[5,3]
40   FOR I=1 TO 5
60   FOR J=1 TO 3
80   READ X[I,J]
100   NEXT J
120   NEXT I
140   DATA 20000,.1,.02
160   DATA 35000,.2,.04
180   DATA 45000,.3,.07
200   DATA 50000,.4,.12
220   DATA 55000,.5,.18
240   READ W0,W1,A0,R,Y0,Y1,Y2,G0,G1,C0,C1,C2,C3,C4,C5
260   READ D0,D1,D2,D3,D4,T0,T1,T2,T3,A1,E
280   O0=0
300   O1=0
320   S0=0
340   S1=0
360   S0=A0+Y0
380   S1=A0+Y0
400   S1=S1+R
420   IF Y1<100 THEN 480
440   S0=S0+Y1-100
460   S1=S1+Y1-100
480   IF G1>G0 THEN 540
500   S0=S0+G0-G1
520   S1=S1+G0-G1
540   IF C0<C1+C2 THEN 600
560   S0=S0+(C0-C1-C2)/2
580   S1=S1+(C0-C1-C2)/2
600   IF C3>C4+C5 THEN 660
620   S0=S0+(C3-C4-C5)/2
640   S1=S1+(C3-C4-C5)/2
660   REM COMPUTE OFFSETS
680   IF C1+C2>C0 THEN 820
700   IF C1+C2-C0>3000 THEN 780
720   O0=O0+C1+C2-C0
740   O1=O1+C1+C2-C0
760   GOTO 820
780   O0=O0+3000
800   O1=O1+3000
820   IF D1>2000 THEN 880
840   Q=D1
860   GOTO 900
880   Q=2000
900   IF D0<.005*(Y0+Y1+Y2) THEN 940
920   Q=Q+D0-.005*(Y0+Y1+Y2)
940   Q=Q+D2/2
960   IF Q<.03*(Y0+Y1+Y2) THEN 1020
980   O0=O0+Q-.03*(Y0+Y1+Y2)
1000   O1=O1+Q-.03*(Y0+Y1+Y2)
1020   O1=O1+D3
1040   O0=O0+D3
1060   IF D4>.25*(Y0+Y1+Y2) THEN 1140
1080   O1=O1+D4
1100   O0=O0+D4
1120   GOTO 1180
1140   O1=O1+.25*(Y0+Y1+Y2)
1160   O0=O0+.25*(Y0+Y1+Y2)
1180   O1=O1+T0
1200   O0=O0+T0
1220   O1=O1+T1
1240   O1=O1+T2
1260   O0=O0+T3
1280   O1=O1+T3
1300   O0=O0+T3
1320   O1=O1+A1
1340   O0=O0+A1
```

Figure C.1. Experimental program ITAXØØ

```
1360  O1=O1+E
1380  O0=O0+E
1400  U0=S0-O0
1420  U1=S1-O1
1440  P1=0
1460  FOR I=1 TO 5
1480  IF U0>X[I,1] THEN 1540
1500  P1=P1+(U0-X[I,1])*X[I,2]
1520  GOTO 1640
1540  P1=P1+X[I,1]*X[I,2]
1560  GOTO 1620
1580  NEXT I
1600  P1=P1+(U0-X[5,1])*X[5,2]
1620  P0=0
1640  P0=0
1660  FOR I=1 TO 5
1680  IF U1>X[I,1] THEN 1740
1700  P0=P0+(U1-X[I,1])*X[I,3]
1720  GOTO 1800
1740  P0=P0+X[I,1]*X[I,3]
1760  NEXT I
1780  P0=P0+(U1-X[5,1])*X[5,3]
1800  PRINT "                    STATE          FEDERAL"
1820  PRINT "============================================="
1840  PRINT " INCOME SOURCES=";
1860  PRINT  USING 1880;S1
1880  IMAGE #,DDDDDDDD
1900  PRINT "  =";
1920  PRINT  USING 1880;S0
1940  PRINT "   ="
1960  PRINT " INCOME OFFSETS=";
1980  PRINT  USING 1880;O1
2000  PRINT "  =";
2020  PRINT  USING 1880;O0
2040  PRINT "   ="
2060  PRINT " NET TAXABLE    =";
2080  PRINT  USING 1880;U1
2100  PRINT "  =";
2120  PRINT  USING 1880;U0
2140  PRINT "   ="
2160  PRINT " INCOME TAX     =";
2180  PRINT  USING 1880;P0
2200  PRINT "  =";
2220  PRINT  USING 1880;P1
2240  PRINT "   ="
2260  PRINT " WITHHOLDING    =";
2280  PRINT  USING 1880;W1
2300  PRINT "  =";
2320  PRINT  USING 1880;W0
2340  PRINT "   ="
2360  P0=P0-W1
2380  P1=P1-W0
2400  PRINT " TAX DUE        =";
2420  PRINT  USING 1880;P0
2440  PRINT "  =";
2460  PRINT  USING 1880;P1
2480  PRINT "   ="
```

Figure C.1.--Continued

LNPR

Program Overview

LNPR provides solutions to linear programming problems by use of the simplex solution technique. Linear programming problems are optimization problems in which a linear objective function of the form

$$Z = C_1x_1 + C_2x_2 + \cdots + c_nx_n$$

is maximized or minimized subject to a set of constraints of the form

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \begin{Bmatrix} < \\ = \\ > \end{Bmatrix} b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \begin{Bmatrix} < \\ = \\ > \end{Bmatrix} b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \begin{Bmatrix} < \\ = \\ > \end{Bmatrix} b_m$$

$$x_j \geq 0 \quad \text{for all } j \ .$$

LNPR provides the optimal values for the decision variables $x_1, \cdots, x_n$ for maximizing the value of Z.

Program Inputs

Inputs to LNPR include the number of decision variables (n); the number of "less than" constraints (ℓ); the number of equality constraints (e); the number of "greater than" constraints (g); the coefficients of the objective function ($c_1,\cdots,c_n$); the coefficients of each constraint, row by row ($a_{11},a_{12},\cdots,a_{1n}$); the "right hand sides" (b_i).

Data for LNPR should begin at line 9900 in the following order:

```
9900 DATA  n, ℓ, g, e
9910 DATA  c1, c2, ··· , cn
9920 DATA  a11, a12, ··· , a1n, b1
9930 DATA  a21, a22, ··· , a2n, b2
  ·            ·
  ·            ·
  ·            ·
     DATA  am1, am2, ··· , amn, bm
```

Program Outputs

LNPR outputs the optimal values of the decision variables $x_1, \cdots, x_n$; the optimal value for the objective function Z; and the array containing coefficients and "right hand side" values for all constraints and objective functions.

The LP problem is solved by either the dual simplex algorithm or the primal simplex algorithm. In the former case, LNPR prints out

```
DUAL SIMPLEX ALGORITHM TO BE USED
```

In the latter case, .

```
PRIMAL SIMPLEX ALGORITHM TO BE USED
```

is reported, and the current value of the LP tableau is printed, in either case.

In the case that any solution is impossible when subject to the input constraints, LNPR prints out:

```
PROBLEM INFEASIBLE
```

In the case that a solution is permissible with a variable unbounded value, LNPR prints out:

```
PROBLEM UNBOUNDED
```

LNPR outputs, at every iteration of the solution algorithm, the values of the matrix which contain the data for the linear programming problem. This matrix is called the linear programming (LP) tableau and is composed initially of

- the values of the coefficients of the constraints,
- the values of the constraint right hand sides
- the values of the coefficients of the objective function, or their negatives

as shown in Figure C.1.

If the LP problem can be solved, LNPR prints out the values of the decision variables $(x_1, x_2, \cdots, x_n)$ which optimize the problem, and reports the constraints which are binding (exactly satisfied).

Program Processing

Contained within LNPR are two algorithms which the program can use to solve the LP problem: the dual simplex method and the primal simplex method. The dual simplex method is chosen by LNPR whenever all of the original objective function coefficients $(c_1, c_2, \cdots, c_n)$ are nonpositive and is employed because it generates fewer artificial variables than the primal algorithm. In all other cases ($c_j > 0$ for some j), the primal simplex method is used.

Whereas the dual simplex algorithm can only be used on problems with nonpositive objective function coefficients, the primal algorithm can only handle constraints with nonnegative right hand sides. Hence, the first step performed by LNPR is to

- determine if the dual simplex algorithm can be used (all $c_j \leq 0$, for all j); if so, all less-than constraints are left as they were input, and all greater-than constraints are converted to less-than constraints by multiplying the coefficients $(a_{i1}, a_{i2}, \cdots, a_{in})$ and right hand side (b_i) by minus one (-1).
- if the primal simplex algorithm must be used (at least one, $c_j > 0$) then all constraints with negative right hand sides ($b_i < 0$) are multiplied through by minus one (-1), turning less-than constraints into greater-thans, and vice versa.

The last preliminary step is to add slack (S_j), surplus (T_j), and artificial (R_j and R'_j) variables to the constraints.

	VARIABLES REPRESENTED BY COLUMNS					
	$x_1 x_2 \cdots x_n$	$S_1 \cdots S_\ell$	$T_1 \cdots T_g$	$R_1' \cdots R_g$	$R_1 \cdots R_e$	Right Hand Sides (b_i's)
1st Constraint						
2nd Constraint						
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
Objective Function Coefficients	$(-)c_1(-)c_2 \cdots (-)c_n$	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	
	n	ℓ	g	g	e	1
	NUMBER OF COLUMNS					

Figure C.2. Linear programming tableau

A matrix A is used to store the coefficients and right hand sides at any stage of the optimization process. Each step of each simplex algorithm transforms $\underline{A}$ until a stopping condition is met. Each transformed constraint and objective function is expressed as an equation and occupies one row of the $\underline{A}$ matrix.

Inequalities of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n < b_i$$

are transformed into the equations of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + S_j = b_i \ .$$

Equalities of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$

are transformed into the equations of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + R_j = b_i \ .$$

Inequalities of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n > b_i$$

are transformed into the equations of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + T_j - R'_j = b_i \ .$$

Each equation above, in addition to the objective function, $Z = c_1x_1 + \cdots + c_nx_n$, is represented row by row in $\underline{A}$ by storing the coefficients for the variables $x_1, \cdots , x_n$; $S_1 , \cdots , S_\ell$; $T_1, \cdots , T_g$; $R'_1, \cdots , R'_g$; $R_1, \cdots , R_e$. The final column of the $\underline{A}$ matrix is used to store the right hand sides of the equations. A diagram of A is shown in Figure C.1.

The primal simplex method for solving a linear programming problem is a cyclic procedure for choosing n of the variables and assigning (or computing) a value for each so that the problem constraints remain satisfied. Those variables so chosen are termed basic variables, and their values are equal to the current right hand sides of the n constraints found in the last column of the A matrix.

Each cycle of the simplex method requires 4 steps. First, the current solution must be tested to see if it is optimal. Second, a new variable must be picked to be basic. Third, a currently-basic variable must be chosen to be excluded from the set of basic variables. And fourth, the matrix must be transformed so that the last column reflects the values of the currently basic variables. The procedure for each step is:

I. Optimality Test: The current objective function coefficients equal, for any non-basic variable, the marginal amount by which the objective function will increase if a variable is picked for inclusion in the next set of basic variables. If all objective function coefficients are negative, then the current solution is optimal. If one or more non-basic variable's coefficient in the objective function is positive, then . . . .

II. Choice of Entering Variable: Choose the variable whose objective function coefficient is most negative, say the variable whose coefficients are found in the k^{th} column.

III. Choice of Exiting Variable: If a coefficient, a_{ik}, in column k is positive, then any increase in the entering variable's value will decrease the value of the basic variable associated with row i. One wishes to increase the entering variable's value to the point where one currently-basic variable's value is reduced to zero.

For every positive entry a_{ik} in column k, compute the ratio of a_{ik} to the right hand side b_i. Choose the smallest ratio, say a_{rk}/b_r, occurring in row r. Row r is the pivot row. If every entry in column k is zero or negative, then the linear programming problem is unbounded.

IV. Pivoting: For the A matrix to properly reflect the coefficients and right hand side values in terms of the new basic variables, it must be transformed by "pivoting" on a_{rk}. Mechanically, this translates into performing elementary row operations on matrix A until the values of the coefficients in the kth row are zero for every row except row R, where a_{rk} equals 1. The elementary row operations that are applied are:

- every element in the rth row is divided by a_{rk}.
- from every row i in A, except the rth is subtracted the rth row multiplied by a_k. That is, for every row element a_{ij}, compute

$$a_{ij} \leftarrow a_{ij} - a_{rj}a_{ik} .$$

Included in this pivoting is every row in A except the rth.

The dual simplex method for solving a linear programming problem is also a cyclic procedure which is the "mirror image" of the primal simplex method. The "goal" of the dual simplex routine is to transform the constraint right hand sides (b_i's) until all are positive. The steps involved are:

I. Feasibility Test: If all the right hand sides (b_i's) are positive, an optimal solution has been reached; otherwise . . . .

II. Choice of Exiting Variable: Choose the most negative right hand side, found in, say, row r. Row r will be the pivot row.

III. Choice of Entering Variable: For every negative coefficient (say, a_{rj}) in row r, compute the ratio of the current objective function coefficient c_j to a_{rj}. Compute a ratio for each negative entry in row r, and choose the smallest ratio, say c_k/a_{rk}. Column k will be the pivot column. If all the entries in row r are positive, the linear programming problem is infeasible.

IV. Pivoting: Pivot on a_{rk} as described above in the primal simplex algorithm.

```
20   DIM A[10,20],Q[10,20],B[10],N[20],R[10]
40   DIM X[20]
60   FOR I=1 TO 10
80   FOR J=1 TO 20
100   A[I,J]=0
120   NEXT J
140   NEXT I
160   B0=0
180   N0=0
200   READ N,L,G,E
220   S=N
240   M=L+E+G
260   FOR I=1 TO M+1
280   FOR J=1 TO N+1
300   Q[I,J]=0
320   NEXT J
340   NEXT I
360   L0=N+1
380   E0=0
400   G0=0
420   FOR J=1 TO N
440   READ A[M+1,J]
460   Q[M+1,J]=A[M+1,J]
480   NEXT J
500   FOR J=1 TO N
520   IF A[M+1,J]>0 THEN 620
540   NEXT J
560   REM ** DUAL SIMPLEX CAN POSSIBLY BE USED
580   D=1
600   GOTO 660
620   REM ** 2 PHASE PRIMAL SIMPLEX MUST BE USED
640   D=0
660   REM
680   REM READ IN LESS THAN CONSTRANTS
700   FOR I=1 TO L
720   FOR J=1 TO N
740   READ A[I,J]
760   Q[I,J]=A[I,J]
780   NEXT J
800   READ R[I]
820   Q[I,N+1]=R[I]
840   IF D=1 OR R[I] >= 0 THEN 980
860   FOR J=1 TO N
880   A[I,J]=-A[I,J]
900   NEXT J
920   R[I]=-R[I]
940   GOSUB 5400
960   GOTO 1000
980   GOSUB 5120
1000   NEXT I
1020   REM READ IN GREATER THAN CONSTRAINTS
1040   FOR I=L+E+1 TO L+E+G
1060   FOR J=1 TO N
1080   READ A[I,J]
1100   Q[I,J]=A[I,J]
1120   NEXT J
1140   READ R[I]
1160   Q[I,N+1]=R[I]
1180   IF D=0 OR R[I] >= 0 THEN 1320
```

Figure C.3. Experimental program LNPRØØ

```
1200  FOR J=1 TO N
1220  A[I,J]=-A[I,J]
1240  NEXT J
1260  R[I]=-R[I]
1280  GOSUB 5120
1300  GOTO 1340
1320  GOSUB 5400
1340  NEXT I
1360  REM READ IN EQUALITY CONSTRAINTS
1380  FOR I=L+1 TO L+E
1400  FOR J=1 TO N
1420  READ A[I,J]
1440  Q[I,J]=A[I,J]
1460  NEXT J
1480  READ R[I]
1500  Q[I,N+1]=R[I]
1520  IF D=1 OR R[I] >= 0 THEN 1620
1540  FOR J=1 TO N
1560  A[I,J]=-A[I,J]
1580  NEXT J
1600  R[I]=-R[I]
1620  GOSUB 5260
1640  NEXT I
1660  FOR J=1 TO N
1700  N[J]=J
1720  NEXT J
1740  FOR I=1 TO M
1760  A[I,S+1]=R[I]
1780  NEXT I
1800  FOR I=1 TO M
1820  IF A[I,S+1]<0 THEN 1880
1840  NEXT I
1860  GOTO 2160
1880  REM PHASE I: ACHIEVE PRIMAL FEASIBILITY
1900  PRINT "DUAL SIMPLEX ALGORITHM TO BE USED"
1920  FOR Q=1 TO 1000
1940  GOSUB 4660
1960  GOSUB 3660
1980  IF R9=-1 THEN 2100
2000  GOSUB 3920
2020  IF C9=-1 THEN 2100
2040  PRINT "PROBLEM INFEASIBLE"
2060  GOSUB 4660
2080  STOP
2100  GOSUB 4220
2120  NEXT Q
2140  A[M+1,S+1]=-A[M+1,S+1]
2160  REM PHASE II: ACHIEVE PRIMAL OPTIMALITY
2180  FOR J=1 TO S+1
2200  A[M+1,J]=-A[M+1,J]
2220  NEXT J
2240  PRINT "PRIMAL SIMPLEX ALGORITHM TO BE USED"
2260  FOR Q=1 TO 1000
2280  GOSUB 4660
2300  GOSUB 3100
2320  IF C9=-1 THEN 2480
2340  GOSUB 3380
2360  IF R9#-1 THEN 2440
2380  PRINT "PROBLEM UNBOUNDED"
2400  GOSUB 4660
```

Figure C.3.--Continued

```
2420   STOP
2440   GOSUB 4220
2460   NEXT Q
2480   FOR J=1 TO N
2500   IF B[I]<0 THEN 2560
2520   NEXT J
2540   GOTO 2600
2560   PRINT "PROBLEM INFEASIBLE"
2580   STOP
2600   PRINT "OPTIMAL OBJECTIVE FUNCTION VALUE IS ";A[M+1,S+1
2620   PRINT
2640   PRINT "VALUES OF DECISION VARIABLES"
2660   PRINT "============================"
2680   PRINT
2700   FOR I=1 TO M
2720   IF B[I]>N THEN 2780
2740   X[B[I]]=A[I,S+1]
2760   PRINT "X";B[I];" = ";A[I,S+1]
2780   NEXT I
2800   FOR J=1 TO S-M
2820   IF ABS(N[J])>N THEN 2880
2840   X[N[J]]=0
2860   PRINT "X";N[J];" = 0"
2880   NEXT J
2900   PRINT
2920   FOR I=1 TO M
2960   FOR J=1 TO N
2980   V=V+O[I,J]*X[J]
3000   NEXT J
3020   IF V#O[I,N+1] THEN 3060
3040   PRINT "CONSTRAINT";I;"BINDING"
3060   NEXT I
3080   STOP
3100   REM ***************************************
3120   REM *  PRIMAL OPTIMALITY TEST:            *
3140   REM *    DETERMINE ENTERING VARIABLE      *
3160   REM ***************************************
3180   C9=-1
3200   Y9=0
3220   FOR J1=1 TO S-M
3240   J=ABS(N[J1])
3260   V=A[M+1,J]
3280   IF V >= Y9 THEN 3340
3300   Y9=V
3320   C9=J
3340   NEXT J1
3360   RETURN
3380   REM *******************************************
3400   REM *   PRIMAL UNBOUNDEDNESS TEST:            *
3420   REM *         DETERMINE EXITING VARIABLE      *
3440   REM *******************************************
3460   R9=-1
3480   Y9=1.E+38
3500   FOR I=1 TO M
3520   IF A[I,C9] <= 0 THEN 3120
3540   V=A[I,S+1]/A[I,C9]
3560   IF V>Y9 THEN 3120
3580   R9=I
3600   Y9=V
3620   NEXT I
```

Figure C.3.--Continued

```
3640  RETURN
3660  REM ****************************************
3680  REM *  DUAL EXITING :                      *
3700  REM *      TEST PRIMAL FEASIBILITY         *
3720  REM ****************************************
3740  R9=-1
3760  Y9=0
3780  FOR I=1 TO M
3800  V=A[I,S+1]
3820  IF V>Y9 THEN 3880
3840  Y9=V
3860  R9=I
3880  NEXT I
3900  RETURN
3920  REM *************************************
3940  REM *  DUAL ENTERING :                  *
3960  REM *          TEST FOR PRIMAL FEASIBILITY *
3980  REM *************************************
4000  C9=-1
4020  Y9=0
4040  FOR J1=1 TO S-M
4060  J=ABS(N[J1])
4080  IF A[R9,J] >= 0 THEN 4180
4100  V=ABS(A[M+1,J]/A[R9,J])
4120  IF V >= Y9 THEN 4180
4140  C9=J
4160  Y9=V
4180  NEXT J1
4200  RETURN
4220  REM ******************************
4240  REM *  PIVOT ON A(R9,C9)          *
4260  REM ******************************
4300  FOR J=1 TO S+1
4320  A[R9,J]=A[R9,J]/A[R9,C9]
4340  NEXT J
4360  FOR I=1 TO M+1
4380  IF I=R9 THEN 4480
4420  FOR J=1 TO S+1
4440  A[I,J]=A[I,J]-A[R9,J]*A[I,C9]
4460  NEXT J
4480  NEXT I
4500  FOR J1=1 TO S-M
4520  IF N[J1]=C9 THEN 4560
4540  NEXT J1
4560  REM EXCHANGE INDICES
4580  T=N[J1]
4600  N[J1]=B[R9]
4620  B[R9]=T
4640  RETURN
4660  REM ***********************
4680  REM *  PRINT TABLEAU       *
4700  REM ***********************
4720  PRINT "          ";
4740  FOR J=1 TO S
4760  PRINT "X";J;
4780  NEXT J
4800  PRINT
4820  FOR I=1 TO M+1
4840  PRINT "X";
4860  IF I=M+1 THEN 4920
```

Figure C.3.--<u>Continued</u>

```
4880  PRINT ABS(B[I]);
4900  GOTO 4940
4920  PRINT " 0     ";
4940  FOR J=1 TO S+1
4960  PRINT  USING 4980;A[I,J]
4980  IMAGE #,DDDD.DD
5000  NEXT J
5020  PRINT
5040  NEXT I
5060  PRINT
5080  PRINT
5100  RETURN
5120  REM ** HANDLE LESS THANS
5140  L0=L0+1
5160  A[I,L0]=1
5180  B0=B0+1
5200  B[B0]=L0
5220  S=S+1
5240  RETURN
5260  REM ** HANDLE EQUALITIES
5280  E0=E0+1
5300  A[I,N+G+L+G0+E0]=1
5320  B0=B0+1
5340  B[B0]=-(N+G+L+G0+E0)
5360  S=S+1
5380  RETURN
5400  REM ** HANDLE GREATER THANS
5420  G0=G0+1
5440  A[I,N+L+G-G0+1]=-1
5460  A[I,N+L+G+G0]=1
5480  N0=N0+1
5500  N[N0]=N+L+G-G0+1
5520  B0=B0+1
5540  B[B0]=-(N+L+G+G0)
5560  S=S+1
5580  RETURN
5590  END
```

Figure C.3.—Continued

OPTM

Program Overview

The OPTIMA program was written to aid economists who wish to find fifth degree polynomials to describe some price curves. The economic model calls for fifth degree polynomials with four local optima as shown in the figure below:

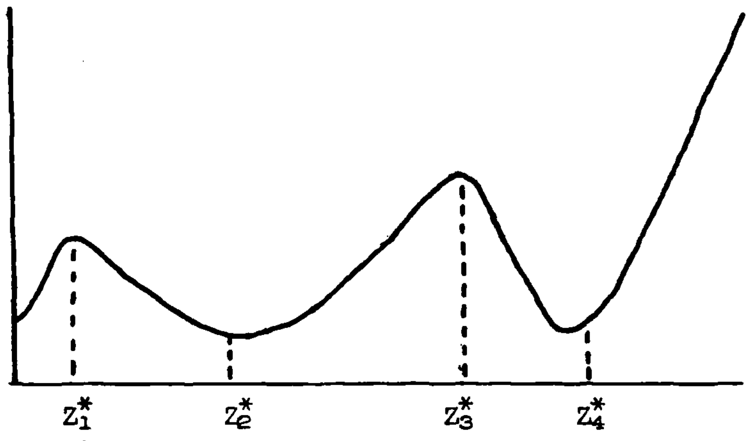


Figure C.4. Sample of fifth degree polynomial

The user inputs the coefficients a_0, a_1, a_2, a_3, a_4, and a_5 of the fifth degree polynomial

$$P(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + a_4 z^4 + a_5 z^5 \ .$$

The program displays the roots of the fourth degree derivative polynomial

$$a_1 + 2a_2 z + 3a_3 z^2 + 4a_4 z^3 + 5a_5 z^4$$

and prints them out. If any of the roots are imaginary, the program stops and prints out the message

```
INPUT POLYNOMIAL DOES NOT HAVE FOUR LOCAL OPTIMA .
```

Program Inputs

Coefficients of P(z) are input as DATA in line 9900, as

```
9900  DATA  a0, a1, a2, a3, a4, a5 .
```

Program Outputs

The program computes the four local optima and prints the answer as

```
LOCAL OPTIMA ARE   3.146   -25.6   39.128   5.543
```

or prints the message

```
INPUT POLYNOMIAL DOES NOT HAVE FOUR LOCAL OPTIMA .
```

Program Processing

The roots of a quartic equation

$$F(z) = b_0 + b_1 z + b_2 z^2 + b_3 z^3 + z^4$$

are found by finding a root y* for the resolvent cubic equation

$$G(y) = y^3 + c_2 y^2 + c_1 y + c_0$$

where

$$c_2 = -b_2$$

$$c_1 = (b_3 b_1 - 4b_0)$$

$$c_0 = -b_3^2 b_0 + 4_2 b_0 - b_1^2 \ .$$

The roots of G(y) are, in turn, found by substituting for y the value $x - c_2/3$ and solving for the roots of

$$H(x) = x^3 + d_1 x + d_0$$

where

$$d_1 = (3c_1 - c_2^2)/3$$

$$d_0 = (2c_2^3 - 9c_2 c_1 + 27c_0)/27 \ .$$

Let $\underline{S} = d_0^2/4 + d_1^3/27$. If F(z) has four real roots then $S < 0$ and $\underline{a}$ root x^* of H(x) is given by

$$x^* = 2\sqrt{-d_1/3}\,\cos(\varphi/3)$$

where

$$\cos\varphi = \frac{-d_0/2}{\sqrt{-d_1^3/27}} .$$

Otherwise, if $S \geq 0$ then a real root exists at the value

$$x^* = \sqrt[3]{\frac{-d_0}{2} + \sqrt{S}} + \sqrt[3]{\frac{-d_0}{2} - \sqrt{S}} .$$

Since $y^* = x^* - c_2/3$ then $x^* - c_2/3$ is a root of $G(y)$, the resolvent cubic equation. Let

$$R' = b_3^2/4 - b_2 + y^* .$$

If R' is negative, then $F(z)$ has imaginary roots and is not suitable. Let

$$R = \sqrt{R'}$$

and

$$D = \begin{cases} \sqrt{3b_3^2/4 - R^2 - 2b_2 + \dfrac{4b_2b_3 - 8b_1 - b_3^3}{4R}} & \text{if } R > 0 \\ \sqrt{3b_3^2/4 - 2b_2 + 2\sqrt{(y^*)^2 - 4b_0}} & \text{if } R = 0 \end{cases}$$

$$E = \begin{cases} \sqrt{3b_3^2/4 - R^2 - 2b_2 - \dfrac{4b_2b_3 - 8b_1 - b_3^3}{4R}} & \text{if } R > 0 \\ \sqrt{3b_3^2/4 - 2b_2 - 2\sqrt{(y^*)^2 - 4b_0}} & \text{if } R = 0 . \end{cases}$$

Then the four roots of $F(z)$ are

$$z^* = -b_3/4 + R/2 \pm D/2 \quad , \quad -b_3/4 - R/2 \pm E/2 .$$

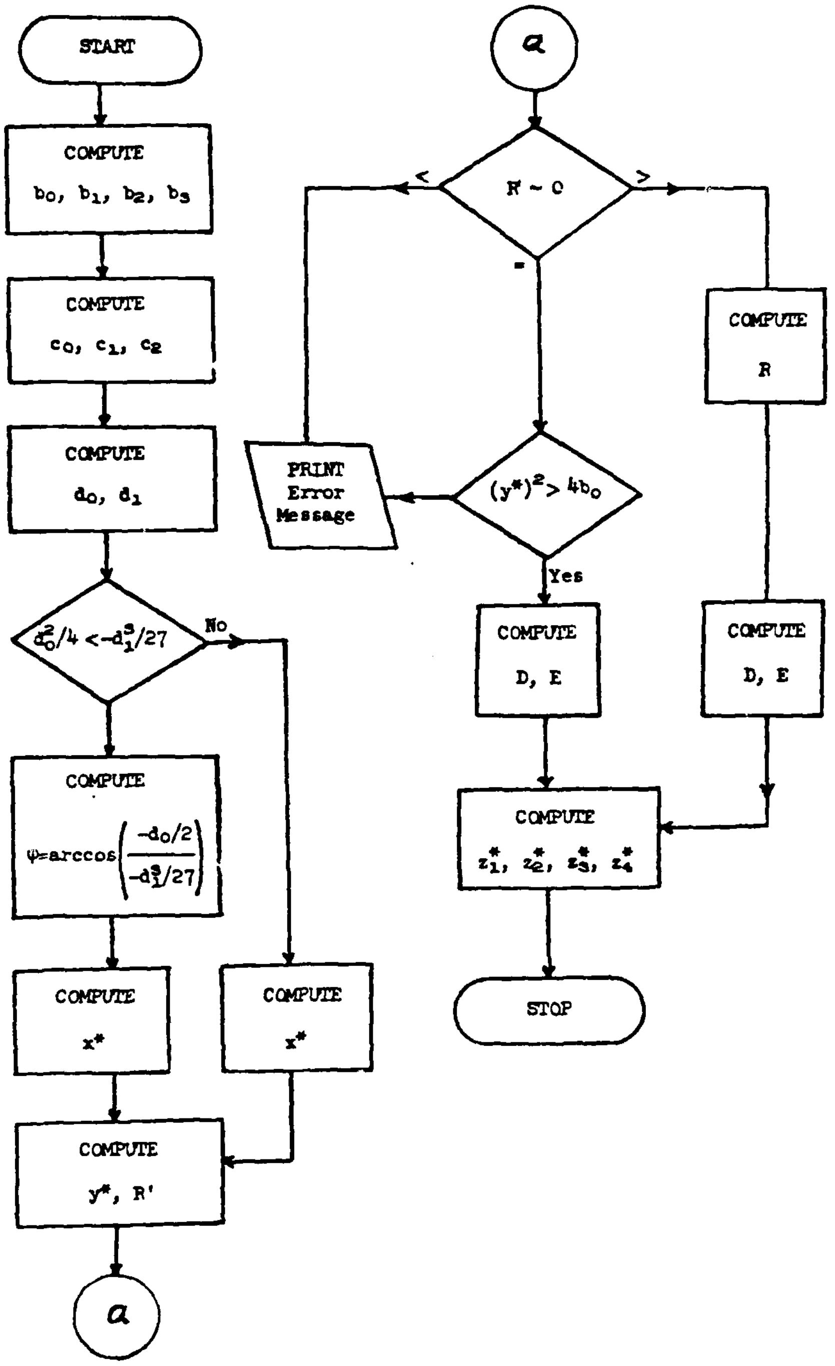


Figure C.5. Flowchart of the OPT'' program

```
10   READ A0,A1,A2,A3,A4,A5
20   B0=A1/(5*A5)
30   B1=2*A2/(5*A5)
40   B2=3*A3/(5*A5)
50   B3=4*A4/(5*A5)
60   C2=-B2
70   C1=(B3*B1-4*B0)
80   C0=B3^2*B0+4*B2*B0-B1^2
90   D1=(3 ** C1-C2^2)/3
100   D0=(2*C2^3-9*C2*C1+27*C0)/27
110   S=D0^2+D1^3/27
120   IF S<0 THEN 180
130   A=(-D0/2+SQR(S))^.3333
140   B=(-D0/2-SQR(S))^.3333
150   X=A+B
160   GOTO 210
170   PRINT "INPUT POLYNOMIAL DOES NOT HAVE FOUR LOCAL OPTIMA
175   GOTO 10
180   LET T1=-D0/(2*SQR(-D1^3/27))
190   LET T=3.14159/2-ATN(T1/SQR(1-T1^2))
200   LET X=2*SQR(-D1/3*COS(T/3))
210   LET Y=X-C2/3
220   R1=B3^2/4-B2+Y
230   IF R1<0 THEN 170
240   R=SQR(R1)
250   IF R#0 THEN 330
260   IF Y^2<4*B0 THEN 170
270   LET P1=3*B3^2/4-2*B2
280   IF P1+P2<0 OR P1-P2<0 THEN 170
290   LET P2=2*SQR(Y^2-4*B0)
300   LET D=SQR(P1+P2)
310   LET E=SQR(P1-P2)
320   GOTO 420
330   REM
340   LET Q1=3*B3^2/4-R^2-2*B2
350   LET Q2=(4*B2*B3-8*B1-B3^3)/(4*R)
360   REM ADJUST FOR NEGATIVE BIAS OF INTERNAL FUNCTIONS
370   Q8=Q1+Q2+.001
380   Q9=Q1-Q2+.001
390   IF Q8<0 OR Q9<0 THEN 170
400   LET D=SQR(Q8)
410   LET E=SQR(Q9)
420   LET Z1=-B3/4+R/2+D/2
430   LET Z2=-B3/4+R/2-D/2
440   LET Z3=-B3/4-R/2+E/2
450   LET Z4=-B3/4-R/2-E/2
460   PRINT "LOCAL OPTIMA ARE ";Z1;Z2;Z3;Z4
470   STOP
```

Figure C.6. Experimental program OPTMØØ

SCNR

Program Description

This program reads in statements written in a fictitious programming language and produces a stream of code pairs corresponding to the symbols which make up the program read in.

The fictitious programming language which is accepted by SCNR is called EASY. EASY is not a line-oriented language; two or more statements may be placed on one line or a statement may be continued on as many lines as necessary. Every statement is ended with a semicolon. There are ten (10) statement types in EASY:

```
<var> := <expression>

DIM <variable>(<bound>, ··· , <bound>)

PRINT <expression>, ··· , <expression>

FOR <variable> := <variable or constant> TO <variable or constant>

WHILE (<condition>)

IF (<condition>) THEN <statement>

READ <variable>, ··· , <variable>

END

GOTO <label>
```

Other rules of EASY are

- Comments can be placed anywhere in an EASY program; a comment begins with the symbol "/*" and continues until the "*/" is found (e.g., " /* THIS IS A COMMENT */").
- Every EASY statement ends with a semicolon (e.g., " X := X + 1;").
- Constants are of two types
  - numeric constants are simple integer numerals or numerals with decimal fractions (e.g., "32" or "47.8").
  - string constants are character strings within single quotes; there is no way to represent a single quote within a string constant (e.g., " 'THIS IS A STRING ' ").

- Statement labels may precede a statement and consist of a proper identifier followed by a colon (e.g., " NEXT: PRINT A;").
- Proper identifiers are variable names and statement labels; an identifier must be one to thirty characters in length, begin with one of the characters {A, B, ··· , C, #, @, $} and be entirely composed of the characters {A, B, ··· , Z, #, $, @, 0, ··· , 9}.
- Blanks are necessary any time a string of characters would be ambiguous. Specifically, to separate
  - a keyword or variable on the left from a numeric constant on the right (e.g., " PRINT 10; ").
  - a keyword from a variable (e.g., "FOR I := 1 TO 10 ; ").

  Any time one blank is used, more than one blank may be used without any change in meaning. This is not true, of course, within comments or string constants.
- Six built-in functions are provided in the EASY language: ABS, SQR, SGN, SIN, COS, AND ATN; each one takes one argument (e.g., " X := SIN(VO#)+ SIN(Y); ").

Each distinct symbol in the language is assigned a symbol class and a subclass number. These assignments are

Symbol	Class Number	Subclass Number
PRINT	1	1
READ	1	2
DIM	1	3
IF	1	4
THEN	1	5
WHILE	1	6
END	1	7
FOR	1	8
TO	1	9
GOTO	1	10
ABS	2	1
SQR	2	2
SGN	2	3
SIN	2	4

Symbol	Class Number	Subclass Number
COS	2	5
ATN	2	6
+	3	1
-	3	2
/	3	3
*	3	4
**	3	5
:=	3	6
<	3	7
>	3	8
<=	3	9
>=	3	10
NOT	3	11
AND	3	12
OR	3	13
:	7	1
;	7	2
,	7	3
(	7	4
)	7	5
<identifier>	6	N/A
<string constant>	5	N/A
<numeric constant>	4	N/A

Program Inputs

SCNR reads EASY program statements as BASIC string constants in DATA statements beginning at line 9900. An example is given below:

```
9900  DATA  "   DIM A(100); "
9910  DATA  "   LET X := X + 1; "
9920  DATA  "   LET A$ := 'STRING' ; "
9930  DATA  "   END;"
```

<u>Program Outputs</u>

SCNR outputs all of the information necessary to reconstruct the input statements. Specifically, SCNR outputs a code pair for each program symbol, a table of identifiers found, and a table of string constants found. Each code pair consists of two parts: the first part is the class number; the second part is either the subclass number (for keywords, operators, functions, and delimiters) or a value (for numeric constants).

A sample of the output expected for the example program above is:

```
IDENTIFIER INFORMATION
========== ===========
NUMBER OF IDENTIFIERS FOUND = 4
IDENTIFIER TABLE
 1             1
 2             4
 5             5
 6             7
IDENTIFIER STRING
ALETXAS

STRING CONSTANT INFORMATION
====== ======== ===========
NUMBER OF STRING CONSTANTS FOUND = 1
STRING CONSTANT TABLE
 1             6
CONSTANT STRING
STRING

SCANNER TOKEN CODE PAIRS
======= ===== ==== =====
 1             3
 0             1
 7             4
 4             1ØØ
 7             5
 7             2
 6             2
 6             3
 3             14
 6             3
```

SCANNER TOKEN CODE PAIRS (contd.)

3	14
4	1
7	2
6	2
6	4
3	14
5	1
7	2
1	7
7	2

SCNR prints out an error message and stops if any of the following limitations are violated:

- the total number of identifiers must not exceed 20.
- the total number of string constants must not exceed 10.
- the combined length of all string constants must not exceed 255.
- the combined length of all identifiers must not exceed 255.
- the total number of symbols must not exceed 100.

Program Processing

SCNR is implemented as a finite state automaton· in which the program progresses from state to state until it reaches an action to take. There are nine states corresponding to distinct "situations" in the decoding of a symbol:

1. initial state
2. currently decoding integer or real numeric constant
3. currently decoding real numeric constant
4. currently decoding keyword or identifier
5. currently decoding identifier
6. currently decoding * or **
7. currently decoding $\left\{\begin{matrix} > \\ < \end{matrix}\right\}$ or $\left\{\begin{matrix} >= \\ <= \end{matrix}\right\}$
8. currently decoding : or :=
9. currently decoding / or /*

There are 14 character types into which the character set is grouped. They are

1. a letter {A, B, ··· , Z}
2. a digit {0, 1, ··· , 9}
3. a National character {#, @, $}
4. a period
5. an asterisk
6. an equal sign
7. an inequality sign
8. a colon
9. a plus or minus sign {+,-}
10. a slash {/}
11. a delimiter {, ; ( )}
12. a quotation mark {'}
13. a blank
14. a fictitious end-of-program symbol {⊢}

SCNR progresses from state to state in decoding the input. A <u>state table</u> contains, for every combination of current state (row) and next character type (column), the next state to assume or action to take. There are eight actions:

ACTION	DESCRIPTION
1	A keyword or identifier has been found; determine which one; if keyword, produce pair;
2	A quotation mark has been found; read incoming characters until another quotation mark is found; store string constant in string table and produce code pair.
3	An operator has been found; determine which one; produce code pair.
4	A numeric constant has been found; determine which one, and produce code pair.
5	End of program; output code pairs and tables.
6	An identifier has been found; if "NOT", "AND", or "OR" proceed to action 3, else ensure identifier is stored in identifier table and produce code pair.
7	A delimiter has been found; determine which one; produce code pair.
8	A comment-begin (/*) has been found; read and ignore all characters until comment-end is found (*/).

SCNR has been implemented by modularizing the program parts into routines. One routine exists for each of the eight actions. Four additional routines are

- GETLINE - gets the next line by reading the next DATA statement; prints out the line together with a line number and removes all but one leading blank.
- GETCHAR - returns the next significant character and its type; returns only the first in a string of blanks; when processing a constant string returns only the character without resetting character type; returns a type of 14 if end of program is detected.
- INIT - initializes all the tables used in the program as well as status variables. These include
  - Search tables K, F, O, D, I, and Q used to search for keywords, functions, operators, delimiters, identifiers, and string constants, respectively. These search tables are used in conjunction with search strings K\$, F\$, O\$, D\$, I\$, and Q\$, where

    X(I,1) and X(I,2)

    contain the first and last character position of the entry in X\$. I and Q are updated during processing whereas K, F, O, and D are static in terms of current active size.
  - Search string BS, containing the legal character set; the B vector is used for determining a character type; if character X resides at position I in BS and $B(J - 1) < 1 \leq B(j)$, then X is in class J.
  - The state table S described above; action entries are stored as negative integers and next-state numbers as positive integers.

  Status Variables

  Q7 = 1 implies processing a string constant,
  0 otherwise

  F7 = 1 implies at least one blank has preceded current char,

  0 otherwise

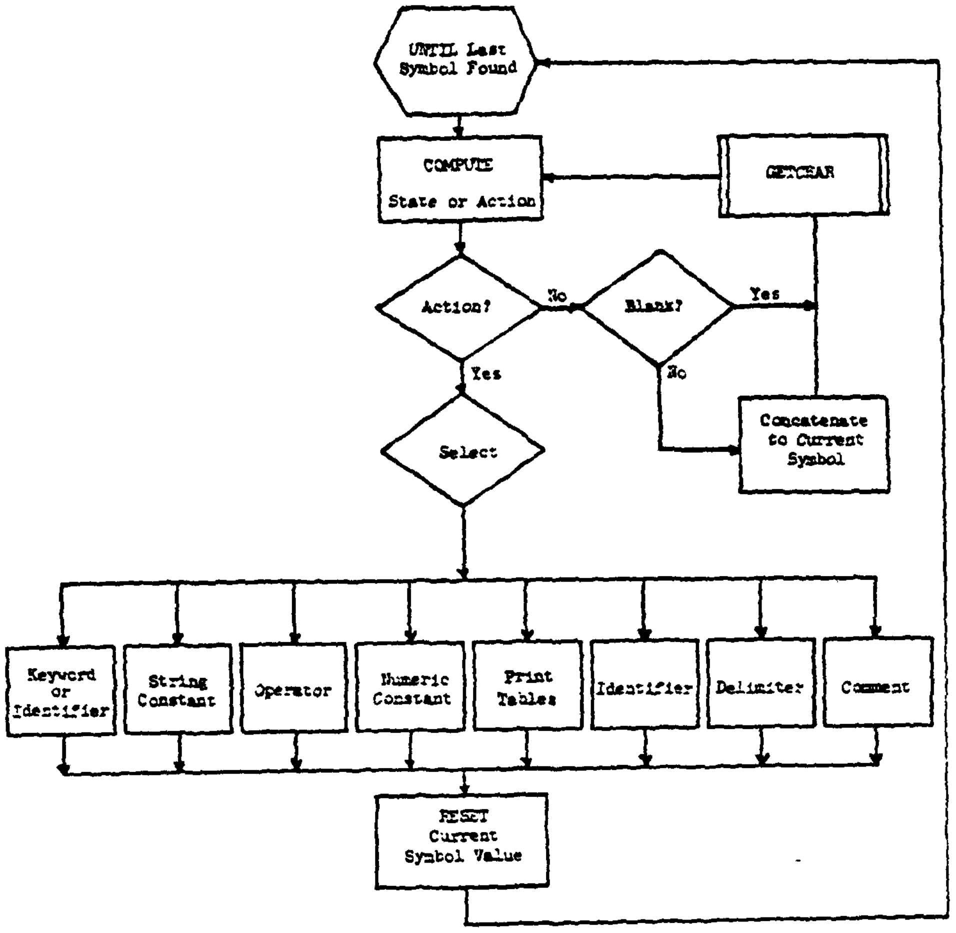


Figure C.7. Flowchart of the SCNR program

```
10   DIM A$[30]
20   GOSUB 2390
30   LET A$=""
40   GOSUB 2030
50   FOR Z8=1 TO 10000
60   LET L=1
70   FOR Z7=1 TO 100000.
80   LET C5=S[L,C]
90   L=C5
100   IF C5 >= 1 THEN 130
110   GOTO -C5 OF 180,410,680,800,1010,1360,1560,1660
120   GOTO 160
130   IF C$=" " THEN 150
133   IF LEN(A$)<30 THEN 140
134   E=6
135   GOSUB 3070
140   LET A$[LEN(A$)+1]=C$
150   GOSUB 2030
160   NEXT Z7
170   REM ***********************
180   REM * KEYWORD OR VARIABLE *
190   REM ***********************
200   FOR X=1 TO K9
210   IF A$=K$[K[X,1],K[X,2]] THEN 240
220   NEXT X
230   GOTO 280
240   C9=C9+1
248   C9=C9+1
250   C[C9,1]=1
260   C[C9,2]=X
270   GOTO 1790
280   FOR X=1 TO F9
290   IF A$=F$[F[X,1],F[X,2]] THEN 320
300   NEXT X
310   GOTO 360
320   IF C9<C6 THEN 328
322   E=6
324   GOSUB 3070
328   C9=C9+1
330   C[C9,1]=2
340   C[C9,2]=X
350   GOTO 1790
360   IF A$="AND" THEN 680
370   IF A$="OR" THEN 680
380   IF A$="NOT" THEN 680
390   GOTO 1360
400   REM ****************************
410   REM * PROCESS STRING CONSTANT *
420   REM TURN ON QUOTE MODE
440   REM ****************************
450   Q9=Q9+1
460   IF Q9 <= 10 THEN 490
470   E=2
480   GOSUB 3070
490   IF Q9>1 THEN 520
500   Q[1,1]=1
510   GOTO 530
520   Q[Q9,1]=Q[Q9-1,2]+1
530   FOR Q8=Q[Q9,1] TO 100
```

Figure C.8. Experimental program SCNR∅∅

```
540   GOSUB 2030
550   IF C$="'" THEN 600
555   IF C#14 THEN 560
556   E=8
558   GOSUB 3070
560   Q$[Q8]=C$
570   NEXT Q8
580   E=3
590   GOSUB 3070
600   Q[Q9,2]=Q[illegible]1
640   C9=C9+1
650   C[C9,1]=5
660   C[C9,2]=Q9
670   GOTO 1790
675   REM *********************
680   REM * PROCESS OPERATOR *
690   REM *********************
700   FOR X=1 TO Q9
710   IF A$=Q$[Q[X,1],Q[X,2]] THEN 730
720   NEXT X
730   IF INT(C5)=C5 THEN 760
740   A$[LEN(A$)+1]=C$
750   GOSUB 2030
760   IF C9<C6 THEN 768
762   E=6
764   GOSUB 3070
768   C9=C9+1
770   C[C9,1]=3
780   C[C9,2]=X
790   GOTO 1790
800   REM *****************************
810   REM * PROCESS NUMERIC CONSTANT *
820   REM *************.***************
830   DIM N$[10]
840   N$="0123456789"
850   V=0
860   P7=LEN(A$)
870   FOR X=1 TO LEN(A$)
880   IF A$[X,X]#"." THEN 910
890   LET P7=X
900   GOTO 950
910   FOR J=0 TO 9
920   IF N$[J+1,J+1]=A$[X,X] THEN 940
930   NEXT J
940   V=V*10+J
950   NEXT X
960   LET V=V/(10^(LEN(A$)-P7+1))
965   IF C9<C6 THEN 970
966   E=6
967   GOSUB 3070
970   C9=C9+1
980   C[C9,1]=4
990   C[C9,2]=V
1000   GOTO 1790
1010   REM *********************************
1020   REM *                               *
1030   REM *    PRINT OUT SCANNER TABLES   *
1040   REM *                               *
1050   REM *********************************
1055   PRINT LIN(2)
```

Figure C.8.--Continued

```
1060  PRINT "IDENTIFIER INFORMATION"
1070  PRINT "********** ***********"
1080  PRINT "NUMBER OF IDENTIFIERS FOUND = ";I9
1090  PRINT "IDENTIFIER TABLE"
1100  FOR X=1 TO I9
1110  PRINT I[X,1],I[X,2]
1120  NEXT X
1130  PRINT "IDENTIFIER STRING"
1140  FOR X=1 TO LEN(I$) STEP 60
1150  PRINT I$[X,X+59]
1160  NEXT X
1170  PRINT
1180  PRINT "STRING CONSTANT INFORMATION"
1190  PRINT "****** ******** ***********"
1200  PRINT "NUMBER OF STRING CONSTANTS FOUND = ";Q9
1210  PRINT "STRING CONSTANT TABLE"
1220  FOR X=1 TO Q9
1230  PRINT Q[X,1],Q[X,2]
1240  NEXT X
1250  PRINT "CONSTANT STRING"
1260  FOR X=1 TO LEN(Q$) STEP 60
1270  PRINT Q$[X,X+59]
1280  NEXT X
1290  PRINT
1300  PRINT "SCANNER TOKEN CODE PAIRS"
1310  PRINT "******* ***** **** *****"
1320  FOR X=1 TO C9
1330  PRINT C[X,1],C[X,2]
1340  NEXT X
1350  STOP
1360  REM ***********************
1370  REM * PROCESS IDENTIFIER *
1380  REM ***********************
1390  FOR X=1 TO I9
1400  IF A$=I$[I[X,1],I[X,2]] THEN 1530
1410  NEXT X
1420  I9=I9+1
1427  GOTO 1440
1430  IF I9<20 THEN 1439
1431  E=5
1432  GOTO 3070
1439  I[I9,1]=I[I9-1,2]+1
1440  I[I9,2]=I[I9,1]+LEN(A$)-1
1450  IF I[I9,2] <= 255 THEN 1480
1460  E=4
1470  GOSUB 3070
1480  I$[I[I9,1]]=A$
1488  I$[I[I9,1]]=A$
1498  C9=C9+1
1500  C[C9,1]=6
1510  C[C9,2]=I9
1520  RETURN
1530  C9=C9+1
1539  C[X,1]=6
1540  C[X,2]=X
1550  GOTO 1790
1560  REM **************************
1570  REM * DELIMETER PROCESSING *
1580  REM **************************
1585  IF INT(C5)=C5 THEN 1590
```

Figure C.8.--Continued

```
1586  A$[LEN(A$)+1]=C$
1587  GOSUB 2030
1590  FOR X=1 TO 09
1600  IF A$=D$[D[X,1],D[X,2]] THEN 1620
1610  NEXT X
1620  IF C9<C6 THEN 1628
1622  E=6
1624  GOSUB 3070
1628  C9=C9+1
1630  C[C9,1]=7
1640  C[C9,2]=X
1650  GOTO 1790
1660  REM **********************
1670  REM * COMMENT PROCESSING *
1680  REM **********************
1690  DIM Z$[1]
1700  GOSUB 2030
1710  FOR X=1 TO 10000
1720  Z$=C$
1730  GOSUB 2030
1740  IF Z$#"*" THEN 1770
1750  IF Z$="/" THEN 1780
1770  NEXT X
1780  GOSUB 2030
1790  A$=""
1795  NEXT Z8
1800  REM*********************
1810  REM*  END OF MAJOR LOOP*
1820  REM*********************
1830  REM *********************************
1840  REM *                               *
1850  REM *  GET THE NEXT LINE            *
1860  REM *                               *
1870  REM *********************************
1880  DIM X$[100],Y$[100]
1890  IF TYP(0)#3 THEN 1920
1900  Y9=-1
1910  RETURN
1920  X$=Y$
1930  READ Y$
1940  X9=X9+1
1950  PRINT TAB(2);X9;TAB(5);Y$
1960  LET Y8=LEN(Y$)
1970  FOR Y9=Y8 TO 1
1980  IF Y$[1,1]#" " THEN 2010
1990  Y$=Y$[2]
2000  NEXT Y9
2010  RETURN
2020  RETURN
2030  REM *********************************
2040  REM *                               *
2050  REM *   GET THE NEXT CHARACTER      *
2060  REM *      IN C$ AND RETURN         *
2070  REM *      CHAR TYPE IN C           *
2080  REM *                               *
2090  REM *********************************
2100  DIM C$[1]
2110  FOR F6=1 TO 1000
2120  GOSUB 2220
2130  IF O7=0 THEN 2150
```

Figure C.8.--Continued

```
2140  RETURN
2150  IF C$=" " THEN 2180
2160  F7=0
2170  RETURN
2180  IF F7#0 THEN 2210
2190  F7=1
2200  RETURN
2210  NEXT F6
2220  IF P1<Y9 THEN 2280
2230  GOSUB 1830
2240  IF Y9 >= 0 THEN 2270
2250  C=13
2260  RETURN
2270  LET P1=0
2280  LET P1=P1+1
2290  LET C$=Y$[P1,P1]
2300  FOR I=1 TO 54
2310  IF B$[I,I]=C$ THEN 2350
2320  NEXT I
2330  E=1
2340  GOSUB 3070
2350  FOR C=1 TO 13
2360  IF I <= B[C] THEN 2380
2370  NEXT C
2380  RETURN
2390  REM ***********************************************
2400  REM *                                             *
2410  REM *   INITIALIZE SEARCH TABLES AND STRINGS      *
2420  REM *                                             *
2430  REM ***********************************************
2440  DIM K[15,2],F[10,2],O[20,2],D[10,2],B[20]
2450  DIM K$[100],F$[100],O$[100],D$[100],B$[100]
2460  F7=0
2470  Q7=0
2480  P1=0
2500  LET K9=10
2510  FOR J=1 TO 2
2511  FOR I=1 TO K9
2512  READ K[I,J]
2513  NEXT I
2514  NEXT J
2520  DATA 1,6,10,13,15,19,24,27,30,32
2530  DATA 5,9,12,14,18,23,26,29,31,36
2540  READ K$
2550  DATA "PRINTREADDIMIFTHENWHILEENDFORTOGOTO"
2560  LET F9=6
2570  FOR J=1 TO 2
2571  FOR I=1 TO F9
2572  READ F[I,J]
2573  NEXT I
2574  NEXT J
2580  DATA 1,4,7,10,13,16
2590  DATA 3,6,9,12,15,19
2600  READ F$
2610  DATA "ABSSQRSGNSINCOSATN"
2620  LET O9=13
2630  FOR J=1 TO 2
2631  FOR I=1 TO O9
2632  READ O[I,J]
2633  NEXT I
```

Figure C.8.--Continued

```
2634  NEXT J
2640  DATA 1,2,3,4,5,7,9,10,11,13,15,18,22
2650  DATA 1,2,3,4,6,8,9,10,12,14,17,21,23
2660  READ D$
2670  DATA "+-/***;=<><=>=NOTANDOR"
2680  LET D9=5
2690  FOR J=1 TO 2
2691  FOR I=1 TO D9
2692  READ D[I,J]
2693  NEXT I
2694  NEXT J
2700  DATA 1,2,3,4,5
2710  DATA 1,2,3,4,5
2720  READ D$
2730  DATA ";;,()"
2740  READ B$
2750  DATA "ABCDEFGHIJKLMNOPQRSTUV*XYZ0123456789*@$,*=<>;+-/;,()'"
2760  B9=13
2770  MAT  READ A[B9]
2780  DATA 26,36,39,40,41,42,44,45,47,48,53,54,55
2790  REM * INITIALIZE SEARCH*
2800  REM *  STRINGS AND      *
2810  REM *    TABLES         *
2820  REM ********************
2830  DIM Q[10,2],I[20,2]
2840  DIM Q$(255),I$(255)
2850  LET Q9=0
2860  LET I9=0
2870  Q$=""
2880  I$=""
2890  REM *******************************
2900  REM * INITIALIZE CODE VECTOR AND   *
2910  REM *    AND STATE TABLE           *
2920  REM *******************************
2930  DIM C(200,2),S[9,14]
2935  LET C6=200
2940  LET C9=0
2950  MAT  READ S[9,14]
2960  DATA 4,2,5,3,6,-3.1,7,8,-3.1,9,-7.1,-2.1,-5
2970  DATA -4,2,-4,3,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4
2980  DATA -4,3,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4
2990  DATA 4,5,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
3000  DATA 5,5,5,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6
3010  DATA -3,-3,-3,-3,-3.1,-3,-3,-3,-3,-3,-3,-3,-3,-3
3020  DATA -3,-3,-3,-3,-3,-3.1,-3,-3,-3,-3,-3,-3,-3,-3
3030  DATA -7,-7,-7,-7,-7,-3.1,-7,-7,-7,-7,-7,-7,-7,-7
3040  DATA -3,-3,-3,-3,-8.1,-3,-3,-3,-3,-3,-3,-3,-3,-3
3050  GOSUB 1830
3060  RETURN
3070  REM **************************************
3080  REM *                                    *
3090  REM *    ERROR DIAGNOSTIC GENERATOR      *
3100  REM *                                    *
3110  REM **************************************
3120  PRINT TAB(P1+7);"^"
3130  PRINT
3140  PRINT "ERROR: ";
3150  GOTO E OF 3160,3180,3200,3220,3240,3260,3300
3160  PRINT "ILLEGAL CHARACTER"
3170  GOTO 1010
```

Figure C.8.--Continued

```
3180  PRINT "TOTAL NUMBER OF STRING CONSTANTS > 10"
3190  GOTO 1010
3200  PRINT "TOTAL LENGTH OF ALL STRING CONSTANTS > 255"
3210  GOTO 1010
3220  PRINT "TOTAL LENGTH OF ALL IDENTIFIERS > 255"
3230  GOTO 1010
3240  PRINT "TOTAL NUMBER OF IDENTIFIERS > 20"
3250  GOTO 1010
3260  PRINT "TOTAL NUMBER OF SYMBOLS > 200"
3270  GOTO 1010
3300  PRINT "PROGRAM ENDS WITH OPEN COMMENT OR QUOTE"
3310  GOTO 1010
```

Figure C.8.--Continued

APPENDIX D

THE CONSTRUCTION OF EXPERIMENTAL TEST DATA SETS

White-box and black-box test data sets were constructed for each of the four experimental programs. A black-box test data set was developed by considering what different processes were identified by a program specification; data were then created to cause one class of program output to occur. Within a black box data set was also included data combinations to test the assumptions and restrictions cited in the specification.

The specific development process for each program's black-box data sets is outlined below:

- ITAX - The input data for two fictitious individuals with complementary and vastly different income and deduction items were used to construct two test data sets.
- LNPR - Six characteristics of linear programming problems were identified: direction of constraint (in)equality, unboundedness, constraint redundancy, infeasibility, negative right hand sides, and negative objective function coefficients. Six combinations of these attributes were grouped and a linear programming problem was constructed corresponding to each of these combinations.
- OPTM - Five test data sets were constructed from problems chosen from a text on the theory of equations.
- SCNR - A sample program was developed which used at least one member of each symbol class. Within the program, statements were included to determine if strings and comments were properly handled. The program was otherwise written to be representative of a "normal" program.

The goal of black-box data construction was to develop test sets representative of typical problems, but sufficiently diverse so as to verify the functions and restrictions set down in the specifications. The goal of white-box data construction is the development of a minimal set of test cases which covers the program graph as completely as possible. It is nearly always infeasible to force the execution of every path sequence, but often possible at least to cover every program branch and important combinations.

The procedure for white-box data set construction was similar for each of the experimental programs. A program flowchart was developed and test data sets calculated to ensure that every program branch was taken. Then, a review of the program code was performed to determine paths of execution which were related through the use of the same variable(s). Data sets which forced execution of important branch pairs were then added. The resulting test sets for ITAX and OPTM appeared little different; because of the low logical complexity both black- and white-box data provided reasonable flowgraph coverage. Black- and white-box test sets for LNPR also appeared somewhat similar since the program processes identified in the black-box procedure were fairly transparently evident in the program logic. The black- and white-box test sets for SCNR appear vastly different; while the black box set represents a comprehensively diverse "typical" program, the white-box set represents a string of symbols designed to force the finite state automaton through its permissible states.

APPENDIX E

EXPERIMENTAL PROGRAM ERRORS AND THEIR CATEGORIZATION

The errors made within each of the four experimental programs were assigned one of three main types: computational, logical, or data handling. Each of these three categories were, in turn, decomposed into subclasses of error types to which every program error was assigned:

- COMPUTATIONAL
  - <u>Missing Computation</u>: One or more statements necessary for proper computation were forgotten.
  - <u>Improper Expression</u>: The computation of a value did not match the specifications.
  - <u>Machine Limitation</u>: An expression, though ostensibly matching the specifications, did not take into account some machine or language limitation which invalidated it.
- LOGICAL
  - <u>Branch/Sequence</u>: Statements which should follow one another during execution did not as a result of improper branching destination or statement permutation.
  - <u>Wrong Boolean Expression</u>: An IF condition was improperly expressed.
  - <u>Missing Logic</u>: A situation which required special handling was overlooked.
  - <u>Redundancy</u>: A statement or statement group was improperly repeated.
- DATA HANDLING
  - <u>Improper Initialization</u>: A variable was initialized improperly through assignment or READ/DATA statements.
  - <u>Wrong Variable Name</u>: The obvious use of the wrong variable name was performed.
  - <u>Subscript/Substring</u>: An improper array index value or substring bound was employed.

The description of errors and designation into error types is given in Tables E.1 through E.4. The breakdown of error type counts by experimental program is shown in Table E.5. Since computational errors are found in expressions and logical errors more prevalent in complex code, it is not surprising that OPTM (high computational content, low logical complexity) and SCNR (low computational content, high logical complexity) exhibit complementary error patterns, while

Table E.1

Error Description: OPTM

Error Number	Statement Numbers	Error Description	Error Types	Error Subclass
1	80	B3 s.b. -B3	Comp	Imp Exp
2	110	80/2 s.b. 80 3/4	Comp	Imp Exp
3	90	3**C1 s.b. 3*C1	Comp	Imp Exp
4	200	/3* s.b. /3)*	Comp	Imp Exp
6	280/290	s.b. permuted	Logic	Wng Br/Seq
7	130/140	s.b. ABS(X) .333*SGN(X)	Comp	Mach Lim
8	470	s.b. GOTO 10	Logic	Wng Br/Seq

<u>Note</u>. For this table and the three tables to follow, the following abbreviations are being used:

Comp	Computational	Wng B Exp	Wrong Branch/ Expression
Wng Op	Wrong Operand	Msg Lgc	Missing Logic
Msg Cmp	Missing Computation	Rdndnt	Redundant
Imp Exp	Improper Expression	D/H	Data Handling
Mach Lim	Machine Limitation	Imp Init	Improper Initialization
Logic	Logical	VBWN	Variable by Wrong Name
Wng Br/Seq	Wrong Branch/ Sequencing	Sub/Sub	Subscript/Substring
	s.b.	should be	

Table E.2

Error Description: LNPR

Error Number	Statement Numbers	Error Description	Error Types	Error Subclass
1	360	N+1 s.b. N	Comp	Imp Exp
2	1180	AND s.b. OR	Logic	Wng B Exp
3	1680/ 1700	s.b. NO=NO+1; N(NO)=J	D/H	Sub/Sub
4	1980	THEN 2100 s.b. THEN 2140	Logic	Wng Br/Seq
5	2480	N s.b. M	D/H	Sub/Sub
6	2500	s.b.···and A(J,S+1)?0···	Logic	Wng B Exp
7	2940	Forgot V=0	D/H	Imp Init
8	4020	0 s.b. 1.E + 38	D/H	Imp Init
9.1	4280/ 4320	A(R9,C9) s.b. out of loop	Comp	Msg Cmp
9.2	4400/ 4440	A(I,C9) s.b. out of loop	Comp	Msg Cmp
10	4520	N(J1) s.b. ABS(n(J1))	Comp	Msg Cmp
11	4560	1 s.b. Z	Comp	Imp Exp
12	2150	Forgot GO TO 2600	Logic	Wng Br/Seq
13	2020	= s.b. #	Logic	Wng B Exp
14.1	3560	3120 s.b. 3620	Logic	Wng Br/Seq
14.2	3520	3120 s.b. 3620	Logic	Wng Br/Seq

Table E.3

Error Description: ITAX

Error Number	Statement Numbers	Error Description	Error Types	Error Subclass
1.1	385	Missing SØ = SØ + Y2	Comp	Msg Cmp
1.2	385	Missing S1 = S1 + Y2	Comp	Msg Cmp
2	400	S1 s.b. SØ	D/H	VBWN
3	600	> s.b. <	Logic	Wng B Exp
4	1260/ 1300	s.b. removed	Comp	Imp Exp
5/6	1460/ 1480	Tax computed	Comp	Imp Exp
	1660/ 1680	Incorrectly		
7	1620	Redundant	D/H	Imp Init
8	680	? s.b. ?	Logic	Wng B Exp
9	1240	O1 s.b. OØ	D/H	VBWN

Table E.4

Error Description: SCNR

Error Number	Statement Number(s)		Error Type	Error Class
1	130-134	Identifier length test	Logic	Msg Lgc
2.1	240-244	Excess code pairs test	Logic	Msg Lgc
2.2	635-37	Excess code pairs test	Logic	Msg Lgc
2.3	1490-1494	Excess code pairs test	Logic	Msg Lgc
2.4	1530-34	Excess code pairs test	Logic	Msg Lgc
3	430	Forgot Q7=1	D/H	Imp Init
4	610-30	Forgot Q7=0	D/H	Imp Init
5	960	+P7)) s.b. -P7 + 1))	Comp	Imp Init
6.1	1425	First identifier check	Logic	Msg Lgc
6.2	1480	First identifier check	Logic	Msg Lgc
7	1520	Return s.b. Goto 1790	Logic	Wng Br/Seq
8	1539-1540	C(X s.b. C(C9	D/H	Sub/Sub
9	1750	Z$ s.b. C$	D/H	VBWN
10	1970	Forgot step-1	Comp	Msg Cmp
11	2490	Forgot X9=0	D/H	Imp Init
12	2750	)'" s.b.)'"	D/H	Imp Init
13	2780	53, 54, 55 s.b. 52, 53, 54	D/H	Imp Init
14	3120	P1 + 5 s.b. P1 + 7	Comp	Imp Exp
15	557	Missing Q9=Q9-1	Comp	Msg Cmp
16	1735-1737	End of symbol test	Logic	Msg Lgc
17	1905	Forgot P1=Y8	Comp	Msg Cmp
18	530	1 to 100 s.b. 1 to 255	D/H	Imp Init
19	2530	36 s.b. 35	D/H	Imp Init
20	2590	19 s.b. 18	D/H	Imp Init
21	2640-2650	18/21,22/23 s.b. 18/20,21/22	D/H	Imp Init
22	2250	C=13 s.b. C=14	D/H	Imp Init
23	2020	Return	Logic	Rdndnt
24	1488	C9=C9+1 unnecessary	Logic	Rdndnt
25	248	C9=C9+1 unnecessary	Logic	Rdndnt
26	1415	I7=1	D/H	Imp Init
27	1427	s.b. deleted	Logic	Wng Br/Seq

Table E.5

Breakdown of Error Type by Program

	ITAX	LNPR	OPTM	SCNR
Computational: Total	4	5	5	5
Missing Comp	2	3	0	3
Imp Exp	2	2	4	2
Machine Limitation	0	0	1	0
Logical: Total	2	7	2	12
Branching/Sequencing	0	4	2	1
Wrong Booleon Expression	2	3	0	0
Missing Logic	0	0	0	8
Redundant Code	0	0	0	3
Data Handling: Total	3	4	0	12
Improper Initialization	1	2	0	10
Variable by Wrong Name	2	0	0	1
Subscript/Substring	0	2	0	1

ITAX and LNPR have more balanced error counts. The propensity to commit data handling errors appears slightly stronger in programs of high logical complexity.

APPENDIX F

CONTENTS OF THE EXPERIMENTAL PACKETS

PARTICIPANT______________________

ACCOUNT NUMBER______________________

PASSWORD______________________

Dear Participant:

I wish to thank you for volunteering to participate in this programming experiment. Your help in this research is not only invaluable to me in my pursuit of a PhD, but more importantly, may help us, the computing community, to better understand the factors affecting software development.

The purpose of this experiment is to collect data concerning the behavior of individuals engaged in program debugging. You will find, enclosed in this packet, specifications for four (4) programs. These programs have been carefully chosen and written to represent many types of software that is developed today. They include

- SCNR - a program which reads in statements in a fictitious programming language, and converts each distinct language symbol into a code.
- OPTM - a program which solves for the roots of a fourth degree polynomial.
- LNPR - a linear programming program which employs the primal and dual simplex algorithms to solve linear optimization problems.
- ITAX - a program which computes the income tax for an individual.

It is expected that you will not be familiar with all of these applications. In fact, you may not be acquainted with any of these problems, but your efforts are just as important as those participants who have had greater experience with these applications.

All of the experimental programs (SCNR, OPTM, LNPR, ITAX) have errors in them. These errors or "bugs" have not been artifically inserted; they occurred during the normal course of program development. IT IS YOUR TASK TO FIND THEM. The number and types of bugs that you find, as well as the activities which lead to their discovery, is the primary information that you will provide for this research.

We realize that everyone has his/her own method of program testing and debugging. However, because the data collected must be comparable from all participants, we ask that your debugging activities be conducted in a pre-set way. During the orientation you will be told the order in which you should select programs for debugging, and the amount of time that you have for each of the four programs. It is important that you

- debug the four programs one at a time,
- in the order that you are instructed to,
- for exactly the amount of time allotted to you - <u>no more, no less</u>.

During the course of the experiment you will <u>always</u> be engaged in one of five (5) activities:

(1) <u>Test Data Development</u>: for those of you who have not been given data to use, you must take time out to construct data to test an experimental program; <u>when you begin and when you end this activity, write the time on your ACTIVITY LOG, included in this packet</u>.

(2) <u>Terminal Work</u>: every time you want to run the latest version of an experimental program, you must log on one of the designated terminals, make whatever changes to the program or data that you have decided upon and run the program; the designated terminals will all be hardcopy devices, so please <u>write your name, participant number, and the time on</u> the terminal paper before logging on; <u>when you begin and when you end this activity, write the time of your ACTIVITY LOG, included in this packet</u>.

(3) <u>Code Review/Error Detection</u>: after leaving the terminal, pick up whatever listings you have asked for from the lineprinter and proceed to debug the program, by considering the listing and results of the program run; <u>when you begin and when you end this activity, write the time on your ACTIVITY LOG, included in this packet</u>.

(4) <u>Error Correction</u>: once you think that you have found an error, terminate the Code Review/Error Detection activity, and begin considering what program changes are necessary to correct the error you have found. Log all intended changes on the PROGRAM MODIFICATION LOG included in this packet. You may switch from "looking for errors" to correcting them many times in one sit-

siting; <u>each switch requires posting a start time and stop time on your ACTIVITY LOG</u>.

The experiment is to be conducted Sunday, January 27, in Bridge Hall 208, on the USC campus. Directions are included in this packet. Orientation will begin promptly at 8:00 A.M. Your early arrival will be enthusiastically appreciated. We expect the experiment to last approximately twelve hours with breaks for resting and meals. Thank you again for your help.

E X P E R I M E N T A L   T I M E T A B L E

8:00 A.M. - 9:00 A.M.	Experimental orientation, distribution of materials
9:00 A.M. - 12:30 P.M.	Experimental Session I
12:30 P.M. - 1:00 P.M.	Lunch break
1:00 P.M. - 4:30 P.M.	Experimental Session II
4:30 P.M. - 5:00 P.M.	Dinner break
5:00 P.M. - 8:00 P.M.	Experimental Session III
8:00 P.M. - 8:30 P.M.	Collection of experimental materials

TERMINAL RULES

(1) You can only LIST, RUN, GET, or EDIT the last version of the program that has been assigned to you.

(2) You are not permitted to PURGE any program or file.

(3) You are not permitted to make any program changes that were not listed on your program modification log when you sat down at the terminal.

(4) You must rename any modified program to the next higher version (e.g., LNPRO4 becomes LNPRO5), and SAVE before leaving the terminal.

ACTIVITY LOG

Subject:______________

Date:________________

Program:______________

Break	Test Data Development	Terminal Work	Code Review/ Error Detection	Error Correction	Start Time	Stop Time	Comments	Program Version

PROGRAM MODIFICATION LOG

Subject:______________
Date:________________
Program:______________

DELETE	INSERT	MODIFY	COMMENTS	TIME	VERSION BEING MODIFIED

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Study of the Factors Affecting Software Testing Performance and Computer Program Reliability Growth	5. TYPE OF REPORT & PERIOD COVERED Technical
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jeffrey Louis Bahr	8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0733
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Management and Policy Sciences University of Southern California Los Angeles, California 90007	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 042-323
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 434 Arlington, Virginia 22217	12. REPORT DATE 1980
	13. NUMBER OF PAGES 278
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software reliability, software testing procedures, testing theory, reliability growth, software reliability models, estimation of error content.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An exploratory study was conducted to determine what personal, environmental and program-related factors affect the process of discovering errors in computer software. Two prevailing philosophies concerning program debugging are the constructive and descriptive approaches. The constructive view of program debugging is that characteristics of the software, the programmer, and the environment in which they interact largely determine the speed and thoroughness with which program errors will be eliminated. The descriptive

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014 6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

view is that the distribution of error discoveries conforms to probabilistic models of reliability growth.

An experiment was conducted in which twenty-two subjects were given a fixed amount of time to find and correct errors in each of four programs. The programs were designed so that each represented one of four combinations of two levels each of logical complexity and conputational content. These programs were thoroughly debugged by the author and a classification of the types and frequencies of errors was recorded. All errors not related to program design were then reinserted.

For each experimental program, a set of specification-based "black-box" data and a set of program structure-based "white-box" data were developed. Each subject was instructed to use four of these eight input data sets in conformance with the experimental design.

During the course of the experiment, each subject partially debugged the four experimental programs in preassigned order. Subjects recorded the duration of each debugging activity and the times at which program errors were calculated, measured by the number of errors found, as well as the distribution and orderings of error discovery times.

The subject group exhibited a wide range of ages, experience and education. Analyses indicated that a reasonable amount of recent programming experience was more important than any other determinant of debugging performance.

An analysis of the errors discovered by the subject group indicated that logical and data-handling errors were less frequently found than computational errors. Moreover, the visibility with which a program error disagreed with the program specifications was directly related to its errors discovery frequency. The wide range of error discovery frequencies confirms that some errors were inherently more difficult than others. This conjecture was also supported by a Chi-square analysis of discovery frequencies, and the independence between error and difficulty and subjects was confirmed by a test of rank concordance among subjects.

An analysis of variance using logical complexity level, computational content and test data type as factors showed only the first two factors as having a significant effect on error discovery. A significantly negative interaction effect of these factors pointed out the inadequacy of the program-characteristic metrics employed.

Maximum likelihood estimates of subjects' proficiencies and errors inherent arrival rates were calculated to test whether or not discovery times were exponentially distributed. Discovery times as a set exhibited a significantly decreasing discovery rate over time, in contradiction to the assumptions of the Helinski-Moranda and Schick-Wolverton software reliability models. A distributional model was developed and tested in which error discovery times are modeled as order statistics on an underlying Pareto distribution. While all three models overestimated the actual number of errors in each experimental program, only the proposed model can readily accommodate the assumption of non-constant discovery rates for individual errors.

ATE
LMED